MAREK JERSÁK

Compositional Performance Analysis for Complex Embedded Applications

Dissertation, 2005

Institute of Computer and Communication Network Engineering
Department of Electrical Engineering and Information Technology
Technical University of Braunschweig
Braunschweig, Germany

Compositional Performance Analysis for Complex Embedded Applications

Von der Gemeinsamen Fakultät für Maschinenbau und Elektrotechnik
der Technische Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung der Würde

eines Doktor-Ingenieurs (Dr.-Ing.)

genehmigte Dissertation

von: Marek Jersak

aus: Bern, Schweiz

eingereicht am: 28. Mai 2004

mündliche Prüfung am: 24. Juni 2004

Referent: Prof. Dr.-Ing. Rolf Ernst

Referent: Prof. Dr.-Ing. Lothar Thiele

Vorsitzender: Prof. Dr.-Ing. Harald Michalik

2005

Compositional Performance Analysis for Complex Embedded Applications

MAREK JERSÁK
Institute of Computer and Communication Network Engineering
Department of Electrical Engineering and Information Technology
Technical University of Braunschweig
Braunschweig, Germany

Abstract

Performance verification is key during the design of embedded systems. It must be ensured that a system meets all performance constraints, in particular hard real-time constraints. The system must also be optimized for cost, size, power-consumption and flexibility to succeed in the market. This requires evaluating the performance impact of each design decision.

Performance verification is challenging, in particular when a large numbers of tasks is implemented on a communication-centric heterogeneous multi-processor architecture with dynamic task and communication scheduling. Function- and subsystem-integration introduce many performance dependencies which are extremely hard to track. Simulation and test are increasingly insufficient for reliable performance verification, since identification of all performance corner-cases and of stimuli sequences to exercise them is unrealistic for complex systems. They are also very time-consuming and require executable code, making them poorly suited for early design-space exploration.

A promising alternative is formal system-level performance analysis. It calculates best-case and worst-case performance bounds and thus guarantees full corner-case coverage, making it a *reliable* approach to performance verification. It requires neither executable code, nor a large test-bench. Instead, only models of those system properties that influence performance are needed, as well as models for the possible timing of task activations. Estimates for those relevant properties and timing information can be obtained early in the design, and can be refined as design progresses. Performance analysis also runs dramatically faster than simulation, making it ideal for design-space exploration.

Recently, our group has developed a compositional approach which enables performance analysis for heterogeneous multi-processor architectures. The novelty of this approach lies in its scalability and its support

for subsystem-integration, which are required for large systems. However, the approach uses a simple application model where one activation of a task depends on exactly one execution of exactly one predecessor task. In realistic embedded applications, a task may require a different amount of data per execution than produced by a predecessor task, leading to *multi-rate systems*. Communication may also be conditional, leading to *execution-rate intervals*. Furthermore, a task may consume data from *multiple task inputs*. Then, task activation timing is a function of the possible arrival timing of all required input data. Applications may also exhibit *cyclic task dependencies*. Any useful system-level performance analysis framework must be able to handle the complexity of real-world applications.

In this work, transformations are developed between the variety of task dependencies that are found in complex embedded applications, and the possible timing of activating events at the implementation level, as required by our compositional performance analysis approach. Specifically, it is shown how to calculate activating event timing in the presence of data rate transitions (with fixed rates and rate intervals) and multiple activating inputs (AND- or OR-concatenated). This includes the analysis of cyclic task dependencies (e. g. in a control loop). If buffering of communicated data is required, then the required buffer sizes as well as minimum and maximum buffering delay are also calculated.

As a further extension, the type of communicated data is captured and considered during analysis, since it can heavily affect processing times and communication load, leading to substantially tighter analysis bounds. Phase information between different task activations can also narrow possible performance bounds, and is considered as well. Finally, to demonstrate a link to real-world design-flows, it is shown how to apply the presented methodology to designs specified in the industry-standard modeling tool Simulink. This additionally requires relaxing the perfectly synchronous Simulink model of computation into a dataflow representation.

In summary, the methodology presented in this thesis allows to apply compositional performance analysis to obtain tight performance bounds for *complex applications* that are mapped onto heterogeneous multiprocessor architectures. Due to analysis speed, the methodology also allows rapid design-space exploration to optimize the implementation of realistic embedded applications while satisfying all critical timing constraints. The presented methodology has been implemented in the system-level performance analysis framework SymTA/S.

Kurzfassung

Moderne eingebettete Systeme führen eine Vielzahl unterschiedlicher Anwendungen auf heterogenen, programmierbaren Multiprozessorsystemen aus. Beispiele sind Multimedia-Plattformen oder verteilte Regelungssysteme im Automobil. Eingebettete Anwendungen müssen häufig in Echtzeit ausgeführt werden. Während des Entwurfs muss daher die Performanz, insbesondere das Echtzeitverhalten, der Systemimplementierung verifiziert werden. Performanzverifikation auf Systemebene ist eine Herausforderung für komplexe Systeme, in denen eine Vielzahl kommunizierender, gemischt transformativ-reaktiver Hardware- und Softwareprozesse auf spezialisierten Multiprozessorsystemen mit verschiedenen Bussen samt Echtzeitbetriebssystemen und arbitrierenden Busprotokollen ausgeführt wird.

Stand der Technik ist simulationsbasierte Performanzverifikation. Aufgrund der Vielzahl von Abhängigkeiten ist es jedoch praktisch unmöglich, alle kritischen Performanzrandfälle zu simulieren. Performanzsimulation ist daher potentiell unzuverlässig. Sie ist auch sehr zeitaufwändig und benötigt ausführbaren Code, und daher wenig geeignet zur frühen und schnellen Entwurfsraumexploration. Eine viel versprechende Alternative ist die so genannte Performanzanalyse. Hierbei werden kürzeste und längster Antwortzeiten von Prozessen berechnet, basierend auf bestimmten Prozesseigenschaften, Schedulingeigenschaften, sowie unter Berücksichtigung aller möglichen Aktivierungszeitpunkte mittels so genannter Ereignismodelle, z.B. periodisch mit einer maximalen Abweichung (Jitter), oder sporadisch mit einem Mindestabstand. Dieses Vorgehen hat attraktive Vorteile: Die berechneten zeitlichen Grenzen sind zuverlässig; performanzrelevante Eigenschaften können zwecks frühzeitiger Exploration schon vor der Implementierung geschätzt werden; Analyse läuft gewöhnlich sehr schnell. Kürzlich wurde am Institut für Datentechnik und Kommunikationsnetze ein kompositionales Verfahren entwickelt,

mit dem Performanzanalyse für beliebig komplexe und heterogene Architekturen anwendbar wird.

Problemdefinition: Unser kompositionaler Ansatz zur Performanzanalyse ist bisher nicht für reale eingebettete Anwendungen mit komplexen Prozessabhängigkeiten anwendbar. Dies liegt daran, dass bisher ein sehr einfaches Aktivierungsmodell verwendet wird, bei dem das Beenden eines Prozesses sofort zur Aktivierung eines abhängigen Prozesses führt. In realen Anwendungen findet sich aber eine Vielzahl verschiedener Prozessabhängigkeiten. Ein konsumierender Prozess kann eine andere Menge von Daten pro Aktivierung benötigen als von einem produzierenden Prozess pro Ausführung erzeugt wird. Dies führt zu Multiratensystemen. Daten können auch bedingt erzeugt werden, was zu Datenratenintervallen führt. Weiter kann ein Prozess Daten von verschiedenen Produzenten konsumieren. Dann sind die möglichen Aktivierungszeitpunkte des Prozesses eine Funktion der möglichen Ankunftszeitpunkte aller benötigten Daten. Schließlich ist das Verhalten von Prozessen im Allgemeinen stark abhängig von der Art der empfangenen Daten. Jede in der Praxis verwendbare Performanzanalyse-Methodik muss die Komplexität realer eingebetteter Anwendungen beherrschen.

Entwickelte Lösung: In der eingereichten Dissertation werden Transformationen entwickelt zwischen der Vielzahl verschiedener Prozessabhängigkeiten auf der Anwendungsebene, und Modellen für mögliche Aktivierungszeitpunkte auf der Implementierungsebene, wie sie von existierenden Performanzanalysen benötigt werden. Konkret wird dargestellt, wie man aktivierenden Ereignismodelle berechnen kann unter Berücksichtigung von Datenratentransitionen ohne Intervalle, Datenratentransitionen mit Intervallen, sowie von mehreren aktivierenden Eingängen, die UND- oder ODER-verknüpft sein können. Dies schließt insbesondere die Analyse zyklischer Prozessabhängigkeiten ein, wie sie z.B. in Regelschleifen gängig sind. Falls Daten gepuffert werden müssen, werden die benötigten Puffergrößen und die maximalen, durch Pufferung verursachten Verzögerungen berechnet.

Darüber hinaus wird die Art von kommunizierten Daten sowie zeitliche Korrelationen zwischen Aktivierungszeitpunkten betrachtet und für die Analyse erfasst, da diese einen starken Einfluss auf Ausführungszeiten und Kommunikationslast haben können, und deren Berücksichtigung eine deutlich engere Berechnung von Grenzen der möglichen Systemperformanz ermöglicht. Schließlich wird gezeigt, wie die entwickelte Methodik für Applikationen angewendet werden kann, die mittels des Standardwerkzeugs Simulink entworfen wurden. Hierzu muss in einem zusätzlichen Schritt zunächst das streng synchrone Simulationsmodell von Simulink relaxiert werden.

Zusammenfassend ermöglicht die in der eingereichten Dissertation entwickelte Methodik, existierende zuverlässige Verfahren zur Performanzverifikation auf komplexe, heterogene, gemischt transformativ-reaktive Applikationen anzuwenden, die auf komplexen, heterogenen Architekturen ausgeführt werden. Aufgrund der Analysegeschwindigkeit erlaubt die Methodik eine schnelle Entwurfsraumexploration, um die Implementierung realistischer eingebetteter Anwendungen zu optimieren, und dabei Echtzeitfähigkeit sicherzustellen. Die theoretischen Ergebnisse wurden im Performanzanalyse-Werkzeug SymTA/S umgesetzt.

Acknowledgments

This thesis is the result of my research at the Institute of Computer and Communication Network Engineering (IDA) at the Technical University of Braunschweig, Germany.

I want to express my gratitude to my advisor Professor Rolf Ernst for his trust and the freedom he gave me to find my way; for sharing his visions, for countless fruitful discussions, and for generously funding my travel to so many conferences. I want to thank Professor Lothar Thiele for helping me focus my research, and for agreeing to co-examine this work. I want to thank Professor Harald Michalik for chairing the examination committee.

Many thanks to Dirk Ziegenbein and Kai Richter, my wonderful, fun and challenging colleagues and friends in the SPI-Project. Kai again for our fruitful collaboration in the SymTA/S project, and for sharing the dream about its future. Many thanks to my colleagues Arne Hamann, Rafik Henia and Razvan Racu who joined the SymTA/S-Project and enriched it both scientifically and personally. Thanks to all other researchers and students who I had the pleasure to work with at IDA, the University of Paderborn, ETH Zürich and other universities. Thanks to the IDA staff for providing professional service otherwise only seen in large companies.

I want to thank my parents for their love, support and determination to guide me towards science. Most of all, I want to thank my wonderful wife Silke for our deep love and the joy to pursue our goals together. Her support and understanding for all the effort I invested made it easy to write this thesis while fully enjoying our personal life.

To all you people who influenced it – without you, this thesis would not have been possible!

Contents

Chapter 1

INTRODUCTION

An embedded system is an application-specific computer system which is embedded in a device that itself often does not appear as a computer. Typical devices containing embedded systems are mobile phones, multimedia devices, automobiles or Internet-routers. Usually, the system environment imposes timing constraints on the embedded system. I. e. the system not only must produce the right results, it must produce them at the right time. In this case, we speak of an *embedded real-time system*. For example, a mobile phone must process a certain amount of voice or data packets within a time frame set by a mobile communication standard. Such *hard timing constraints* must be satisfied under all operating conditions. Safety-critical systems are an even more extreme case. For example, an airbag in a vehicle must fire within a guaranteed time interval after impact detection to optimally protect a passenger.

A modern embedded real-time system executes a set of complex, specialized applications on a specialized architecture. Designing such a system is an elaborate and costly task. A particularly challenging design-step is the implementation of an executable function specification, which is target hardware independent, on a real architecture with a limited number of specialized resources and finite hardware speed. At this point, performance becomes a major issue as the implementation has to meet all timing constraints. The design-step during which system performance is validated against timing constraints is called *performance verification*.

Architecture and implementation also need to be optimized for cost, power consumption, extensibility and other aspects. Good architecture and implementation decisions can mean the difference between a profitable product and one that cannot be sold. Ever increasing function

and hardware architecture complexity lead to a huge number of implementation alternatives. Each implementation decision influences the performance of the system. Therefore, the performance impact of each implementation decision needs to be evaluated and compared against performance constraints and optimization goals. This process is often called *design-space exploration*.

Design-space exploration and performance verification have a lot in common. While the goal of design-space exploration is to rapidly estimate the impact of implementation decisions on system performance, in order to identify promising candidates, the goal of performance verification is to validate the performance of a chosen system implementation in detail against performance constraints that are specified in the function specification. Consequently, similar techniques can be applied to address both problems.

1.1 Motivation

System-level performance verification is challenging for complex systems, in particular when a large number of communicating hardware and software tasks are executed on heterogeneous multi-processor/multi-bus architectures with real-time operating systems (RTOSes) and communication via shared busses. The integration of different functions on a heterogeneous architecture introduces a confusing number of run-time interdependencies that heavily affect system performance and can be extremely hard to track [97]. For example, a task may be delayed beyond its deadline due to a transient overload on a resource, which occurred because a burst of data arrived from a different resource. Performance verification has been identified as one of the top 3 system-level design issues in the 2001 International Technology Roadmap for Semiconductors [99].

The key in performance verification is to cover performance corner-cases. A performance corner-case is the smallest or largest value that can be observed for a performance property during regular operation of a system, independent of the operation history. It must fall within the bounds defined by hard performance constraints. During design-space exploration, corner-cases are equally important since these are the cases for which good estimates of system performance are most urgently required.

State of the art in performance verification are cycle-true simulation, as well as test using prototype hardware. These approaches are increasingly insufficient for the reliable verification of hard performance constraints, since identification of all performance corner-cases and of stimuli sequences to exercise them is unrealistic for sufficiently large

and complex systems. Simulation and test are thus potentially unreliable for system-level performance verification. They are also very time-consuming and require executable code, which makes them poorly suited for early design-space exploration.

Formal system-level performance analysis is an interesting alternative to simulation and test for performance verification and design-space exploration. It calculates best-case and worst-case performance bounds and thus guarantees full corner-case coverage, making it a *reliable* approach. Furthermore, performance analysis requires neither executable code, nor a large test-bench. Instead, only models of those system properties that influence performance are needed, as well as models for the possible timing of task activations. Estimates for those relevant properties and timing information can be obtained early in the design, and can be refined as design progresses. Since performance analysis also runs dramatically faster than simulation [109, 50], it is ideally suited for early and thorough design-space exploration.

If performance analysis has so many advantages over simulation, why isn't is widely used? Performance analysis faces three major challenges. Firstly, a performance analysis framework must be able to handle a variety of performance interdependencies resulting from the complexity of todays communication-centric, heterogeneous multi-processor architectures. Secondly, architecture and timing models as well as algorithms used for performance analysis must be sufficiently accurate to calculate performance bounds are both conservative and tight. Thirdly, a performance analysis framework must be able to handle a variety of complex functional interdependencies that are common in state-of-the-art embedded applications.

Recently, our group has developed a compositional approach which enables performance analysis for heterogeneous multi-processor architectures [97, 98]. The novelty of this approach lies in its scalability and support its for subsystem-integration, which are required for large systems. However, the approach uses an simple application model where one activation of a task depends on exactly one execution of exactly one predecessor task.

In realistic applications, a consumer task may require a different amount of data per execution than produced by a producer task, leading to *multi-rate systems*. Communication may also be conditional, leading to *execution-rate intervals*. Furthermore, a task may consume data from *multiple task inputs*. Then, task activation timing is a function of the possible arrival timing of all required input data. Applications may also exhibit *cyclic task dependencies*, which require analysis extensions to accurately consider correlations induced by the cycle.

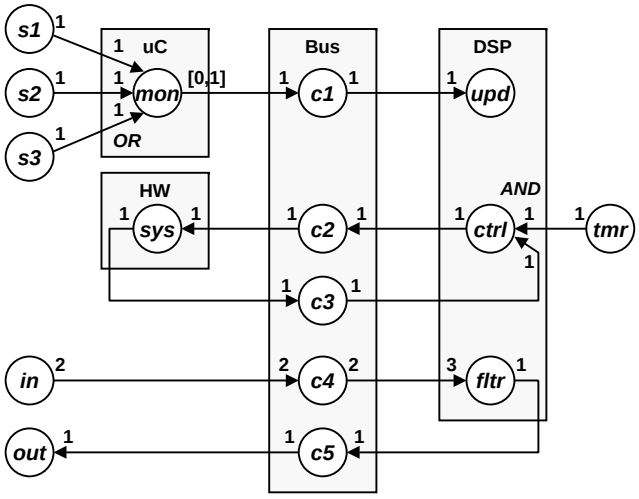


Figure 1.1. Example of an embedded real-time system with a variety of task dependencies

Consider the example in Fig. 1.1. It shows an embedded system consisting of a micro-controller (uC), a digital signal processor (DSP) and dedicated hardware (HW), all connected via a system bus (Bus). A number of computation and communication tasks share theses components, and display a variety of functional dependencies. Task *mon* is an OR-activated task, which reacts to new input data at any of its inputs. Task *ctrl* is AND-activated, and reacts once new input data is available at all of its inputs. Task *ctrl* and *sys* form a functional cycle together with communication channels *c2* and *c3*, which strongly correlates the timing of data produced and consumed by task *ctrl*. Data arrives at the input of task *fltr* in groups of 2 tokens, but the task requires 3 tokens for one execution. Task *mon* produces output data conditionally. All these functional dependencies influence the possible timing of task activations, and consequently system performance. Additionally, non-functional dependencies between tasks due to sharing of the *Bus* and the *DSP* also influence performance. Any useful system-level performance analysis framework must be able to consider both the variety of functional, and the variety non-functional dependencies in complex embedded systems.

Our compositional performance analysis approach can currently cope with the heterogenous architecture in our example, but it cannot be applied due to the complex application structure. On the other hand, applications like this are easily modeled in a variety of system-level languages and tools. Obviously, there is a missing link in system-level design between application models and performance analysis. This thesis pro-

vides a solution how to close this gap, in order to enable compositional performance analysis for real-world embedded applications.

As a further extension, the type of communicated data, which is known from the application model, is captured and considered during analysis, since it can heavily affect processing times and communication load, leading to substantially tighter analysis bounds. Phase information between different task activations can also narrow possible performance bounds, and is considered as well. Finally, to demonstrate a link to real-world application modeling, it is shown how to apply the presented methodology to designs specified in the industry-standard modeling tool Simulink.

1.2 Outline

The remainder of this thesis is organized as follows.

Chapter 2 starts with an overview of the state-of-the-art in system-level design for embedded systems. We emphasize the complexity and heterogeneity of modern embedded applications, and cover the resulting variety of modeling languages and tools. We also illustrate the need for specialized multi-processor target architectures. We then explain why function and subsystem integration leads to complex performance dependencies, in particular for systems containing a large amount of embedded software. These performance problems motivate formal system-level performance analysis. We illustrate the role of performance analysis during system-level design. The chapter concludes with a brief presentation of the SPI model which comprehensively captures application properties for the purpose of system-level performance analysis.

Chapter 3 gives an overview of the state-of-the-art in formal system-level performance analysis. Scheduling analysis, which calculates worst- and best-case response times for tasks sharing a single component, forms the basis of sophisticated performance analysis approaches for multi-processor architectures. We present several of these approaches. Our main focus is on a compositional performance analysis approach which allows to combine different local scheduling analysis techniques to enable performance analysis for arbitrary architectures. A missing link is then identified between realistic application models on one hand, and compositional performance analysis on the other. While realistic application models do not adequately take architecture into account, compositional performance analysis works only for very simple applications.

The following three chapters describe our novel approach to enable compositional performance analysis for realistic applications with a variety of task dependencies. *Chapter 4* focuses on tasks with multiple activating inputs. We argue that AND- and OR-activation are the only

meaningful concatenations of multiple activating inputs. We calculate possible activation timing, as well as required communication buffers and incurred token delay for both types of activation dependencies. The chapter concludes with a brief discussion of combination of AND- and OR-activation.

Chapter 5 focuses on data rate transitions between a producer and a consumer task. We start with transitions between fixed data rates, and then extend our ideas to transitions between data rate intervals, in order to model conditional communication. In both cases, we calculate possible activation timing, as well as required communication buffers and incurred token delay. We then combine the results from this chapter with the results from chapter 4 to enable analysis of tasks with both data rate transitions and multiple inputs. The chapter concludes with a discussion of the required communication buffers.

Chapter 6 shows how to consider cyclic task dependencies during compositional performance analysis. The presented solution for single-rate cycles allows to reuse previous results with only minimal modifications, thus avoiding restricting the set of analyses that can be composed. The chapter concludes with an outlook on analysis of multi-rate cycles.

In *chapter 7*, we apply the results from the previous chapters to analyze the performance of our initial system-on-chip example. We show how a designer is guided through the analysis and performance verification process, and demonstrate the potential of our approach for design-space exploration.

Chapter 8 shows how existing system-level models and analysis techniques can be extended to exploit certain correlations between consecutive task activations. Specifically, we consider sequences of activations of the same task (intra contexts), and possible phases between activations of different tasks (inter contexts). We show that significantly tighter analysis results can be obtained when contexts are considered.

Chapter 9 gives an outlook on integrating system-level performance analysis into a design-flow starting from the industry-standard modeling tools Simulink. The limitations of schedulers synthesizable from Simulink today are discussed first. Then, an improved solution is presented where a Simulink model is first transformed into a suitable task-graph representation, which can then be scheduled more freely, and additionally allows to apply the performance analysis techniques developed in chapters 4 - 6.

Chapter 10 concludes the thesis with a summary and an outlook on future research directions.

Chapter 2

SYSTEM-LEVEL DESIGN

This chapter presents a brief overview over system-level design. It is needed to understand the role of formal system-level performance analysis and its integration into a system-level design-flow, which is the focus of this work. We cannot give a full account in a few pages of the many design-flows that exist. Therefore, we mainly point out key commonalities. We focus on problems that motivate the integration of formal system-level performance analysis into system-level design.

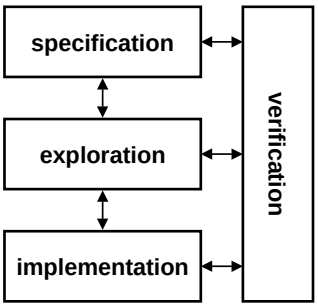


Figure 2.1. Coarse-grain view of a generic system-level design flow

A coarse-grain view of a generic system-level design flow is shown in Fig. 2.1. Starting from a requirements specification, the first system-level design step is to build an *executable specification* that models the intended system functionality [34]. Often, multiple design-teams work on different system functions. Some functions may also be re-used from previous designs (legacy code), or purchased as intellectual property (IP) from a 3rd party. In parallel to function design, a test-bench for system *verification* is developed. The executable specification is combined with

the test-bench in a simulation environment in order to test its correctness. Certain critical functions may also be verified using formal model checking techniques [21]. During this phase, the target architecture is idealized, e. g. by assuming zero or unit computation and communication latencies, perfect parallelism, unlimited buffers etc. Such idealizations are attractive at this stage, since they allow to focus on function design and verification without having to worry about implementation details and implementation verification.

Once the executable specification satisfies functional requirements, the focus shifts to target architecture design and function implementation. The target architecture is usually at least partially given due to the need to use standardized components and evolve previous designs, and constrained by the intended cost, size etc. of each unit. The challenge becomes to integrate all functions on the target architecture under the constraints imposed by a limited number of resources running at finite speeds.

At this stage, implementation verification becomes a key issue. On one hand it must be assured that different subsystems correctly function together. On the other hand, it must be assured that the system meets all performance constraints. The test-bench is re-used for this purpose, together with additional tests. The implementation is also rated with respect to optimization goals. Problems at this stage can often be fixed through implementation changes, but sometimes also necessitate re-design of parts of the function or changes to the target architecture. This is expressed e. g. in the *Y-model* (Fig. 2.2) which is often used in hardware/software co-design [28].

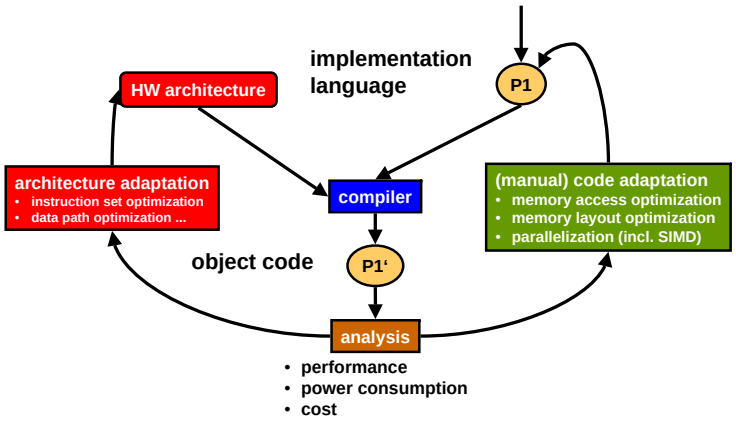


Figure 2.2. Y-model for hardware/software co-design

Implementation changes and implementation verification form several cycles which are iterated as often as necessary to reach an acceptable implementation. Such iterations can be costly, in particular if a problem discovered late in the design can only be fixed through a major functional or architecture change, which then impacts many other parts of the system.

Typically, only *performance* problems have the potential for such a large-scale impact. Functional problems tend to be confined to interfaces between functions or stem from incomplete or contradicting requirements, and usually can be fixed locally. This is because it is well-established design practice to modularize functions such that most interdependencies are localized, and interaction between modules is kept to a well-defined minimum. On the other hand, there is no simple approach to 'performance modularization' or budgeting, since performance interdependencies are introduced between functionally independent tasks due to resource sharing.

Performance requirements are a key motivator for specialized embedded architectures[1]. Good architecture and implementation decisions can mean the difference between a profitable product and one that cannot be sold. Therefore, it is desirable to thoroughly evaluate architecture and implementation alternatives to arrive at an attractive solution. This process is generally called *design-space exploration*. Obviously, design-space exploration is limited if a system has to be fully implemented before its performance can be evaluated. Therefore, industry strives to work with simulation-based *virtual prototypes* as soon as possible, which allow major changes relatively quickly and at relatively low cost. However, turnaround-times are still in the range of hours to days.

2.1 Executable Specifications

An executable specification is an executable model of the desired system function. In this section, typical characteristics of executable specifications are described.

From a system-level perspective, an executable specification is divided into multiple communicating functions. The *coordination*, meaning the communication structure as well as activation dependencies between functions [123] can be complex and consequently are best formalized. A formal representation of these system-level dependencies is called a model of computation (MOC) [100]. A MOC can be seen as the 'laws of physics' which govern the interaction of elements of the model [60].

[1] A second major motivator, which is not considered in this thesis is power consumption.

There is no single best MOC for all modeling problems. Instead, several established MOCs exist that are tailored towards a specific application properties (e. g. predominantly reactive vs. predominantly transformative, data or control dominated) [60, 123]. Lee has identified a set of 'basic' MOCs with fundamentally different key properties [64, 60].

Different research and commercial modeling tools are available which either implement these basic MOCs directly, or as a variation or combination. Modeling tools are typically graphical and often tailored towards specific application domains (e.g. signal processing, communication protocols, control systems). Major benefits of mature, specialized tools include quick design entry, availability of large function-component libraries, excellent support for simulation, debugging and optimization, as well as automatic code-generation.

We will now take a closer look at some important MOCs and tools that implement them. The focus will be on the variety of functional dependencies that can be modeled, as well as on suitability for implementation on complex multi-processor architectures. A more extensive overview can be found in [123].

2.1.1 Dataflow Process Networks

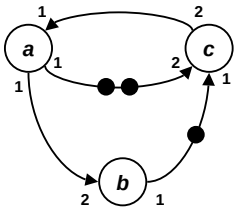
Dataflow process networks [63], a special case of Kahn process networks [55], operate on streams of data. Processes communicate via FIFO-buffers with unrestricted size, using non-blocking write and blocking read semantics. Processes are decomposed into sequences of indivisible quantum of computation called actor *firings*. During each firing an actor consumes input *tokens* and produces output tokens, where a token is considered an elementary unit of data. *Activation rules* specify the type and number of data tokens which have to be available at the actor inputs in order to allow an actor to fire. A key property of dataflow process networks is that an actor firing is solely dependent on the availability of data. This implies that dataflow process networks are *un-timed*, meaning that nothing is specified about points in time at which firings occur. Only valid firing *sequences* can be determined for dataflow process networks.

Dataflow process networks expose the parallelism in an application. Any actor with sufficient input data can execute in parallel to any other actor with sufficient input data. Due to FIFO-communication, the order in which actors execute, including actors interrupting each other, does not affect the calculated results, i. e. dataflow process networks are determinate in the sense that internal and output streams depend only on input streams [63]. These properties greatly simplify mixed hardware/software implementation on parallel architectures, since re-

sults cannot change due to different execution times of parallel actors, or due to a particular processor- or bus-scheduling strategy. As we will see below, several other MOCs do not have this benefit.

Token rate transitions are allowed between two actors, leading to different execution rates of these actors (multi-rate system). An actor can also have *multiple inputs*, in which case the required number of tokens must be available at each actor input for one firing (*AND-concatenation*). Actors can also form *functional cycles*. In cycles, a sufficient number of initial tokens must be available on some cycle edge to avoid deadlock.

Several specializations of dataflow process networks have been developed. Of particular importance is *synchronous dataflow (SDF)* [61], where the number of tokens consumed at each input and produced at each output by each actor per firing is fixed and specified a priori. This allows to determine a static firing sequences which returns the SDF graph into its initial state. Such a firing sequence can be repeated in a loop to statically schedule an SDF graph operating on a stream of data [62]. Fig. 2.3 shows a simple SDF graph and a valid sequential firing sequence that returns the graph into its initial state. Most of the existing work on SDF scheduling focuses on optimizing static schedules for parallel execution, required sizes of FIFO-buffers, or end-to-end latency (e. g. [82, 89]).



		tokens on edges after firing			
step	firing	$a \rightarrow b$	$b \rightarrow c$	$c \rightarrow a$	$a \rightarrow c$
0			●		●●
1	c			●●	
2	a	●		●	●
3	a	●●			●●
4	b		●		●●

Figure 2.3. A simple SDF graph and a valid sequential firing sequence that returns the graph into its initial state

In the *cyclo-static dataflow (CSDF)* model [27], the number of consumed and produced tokens by an actor may vary cyclically. For example, an actor may consume one token at its odd firings and consume three tokens at its even firings. This still yields a statically determinable firing sequence as for an CSDF network a corresponding SDF network can be created by unfolding. A further generalization is *boolean dataflow (BDF)* [12] where the numbers of consumed and produced tokens depends on the value of a token read from a dedicated control input. While

this results in Turing completeness, it also causes a BDF process network to be in general not statically schedulable.

2.1.2 Globally Synchronous Models

In globally synchronous models, parallel actors are assumed to execute at *exactly* the same points in time. Synchrony is enforced by perfect clocks. Execution and communication are usually timeless. Application topologies are not restricted and can include actors with multiple inputs and outputs, as well as cycles. Simulink [74] allows multiple clocks with different clock-speeds in one system. All actors connected to the same clock are activated periodically at the same time, independent of a change at actor inputs, and always produce output values. We will consider Simulink in detail in chapter 9. In synchronous/reactive models [37] all processes simultaneously react to changes at their inputs and may produce new output values (conditional communication). A change at any input of a multi-input actor causes a reaction (*OR-activation*). One example are hierarchical finite state machines (HFSMs), e. g. Harel's StateCharts [38], where the reaction of parallel FSMs to an input event is instantaneous and infinitely fast.

A major problem with global synchrony is that it does not scale well to distributed architectures, since enforcing global synchrony across distributed resources makes the system inefficient [20]. Additionally, such a system is often globally sensitive to small local changes. On the other hand, the timing in a properly implemented distributed synchronous system is highly predictable, and consequently popular for safety-critical applications. One example for a distributed synchronous system is the time-triggered protocol (TTP) [57] currently considered as a candidate for system-level communication in automotive networks for applications like steer-by-wire and break-by-wire.

2.1.3 Globally Asynchronous Models

One workaround around the limitations of a fully synchronous large system is a locally synchronous, but globally asynchronous system. This paradigm has been implemented e. g. in the *Codesign Finite State Machines (CFSM)* [18] used in the Polis codesign environment [6]. A disadvantage is that the system behavior depends on the arrival order of events between synchronous regions, and is thus fundamentally implementation-dependent [123]. The same is true for system designed in the *Specification and Description Language (SDL)* [45], a language widely used in the design of telecommunication protocols. In SDL, processes representing extended FSMs communicate via unbounded FI-

FOs. However, different to dataflow, all senders write into the same unbounded input queue. The resulting non-deterministic system behavior is usually acceptable, since it reflects the nature of typical telecommunication systems [103].

2.1.4 Multi-Language Design

Embedded systems often combine aspects from different application domains. For example, a state-of-the-art mobile phone needs to integrate various transformative signal processing functions, multiple reactive communication protocols, and an interactive user interface. These fundamentally different application properties encourage multi-language design, since a good model for one aspect is often awkward for another [19]. Two basic approaches can be distinguished [123]. In the *compositional* approach, different MOCs are integrated hierarchically. A prominent research tool supporting this approach is Ptolemy [117].

A special case is the unified modeling language (UML) [70] which covers many different semantics but currently has little simulation, verification and optimization support. UML is thus positioned somewhere between requirements specification and executable models.

The second approach is *co-simulation*. Different system parts are designed with different tools, and then co-simulated using co-simulation interfaces that most tools provide. For example, in [10] an efficient co-simulation approach for Matlab on one hand, and SDL on the other is presented. The simulation results from one tool can also be used in form of characteristic values during a simulation in a different tool. This approach makes sense if some level of detail, often the granularity of time, is significantly different in both simulations, and co-simulation thus would be either extremely inaccurate, or extremely time-consuming.

2.1.5 Commercial Tools

All widely-used MOCs are supported by one or more commercial tools. Simulink/StateFlow [74, 75] by The MathWorks implements both continuous-time and perfectly synchronous MOCs. The tool is often used for the modeling of control systems and signal-processing systems. A transformative Simulink actor can internally be described as a StateFlow FSM (composition). Simulink will be discussed in detail in chapter 9.

Tau [108] by Telelogic implements the language SDL (section 2.1.3), and is widely used for the specification of telecommunication protocols. CoCentric System Studio [106] by Synopsys implements various dataflow models of computation, and allows hierarchical composition with un-timed FSMs. It is used for the design of signal-processing sys-

tems. StateMate [42] by I-Logix implements synchronous parallel FSMs (Harel's StateCharts [38]), and as such is used for complex, local control algorithms.

2.1.6 System-Level and Implementation Languages

All modeling tools offer code-generation into implementation languages, in particular C for functions marked for software implementation, and VHDL or Verilog for functions that are to be implemented in hardware. C++ and Java are alternatives to C for some application domains. A modern alternative are system-level languages, which are both suited for software and hardware implementation and thus allow the designer to delay this decision. Two prominent examples are SystemC [83], which is implemented as a C++ library, and SpecC [34], which is an extension of C.

In many designs, graphical model-based approaches are never used. Instead, the design starts directly with system-level or implementation languages. In other designs, at least part of the function is specified in these languages, either as a means to manually optimize the function, or because legacy code in such a language needs to be integrated. Specifications in implementation languages offer maximum flexibility and optimization potential, since these languages do not restrict the designer in any way beyond the basic language semantics. However, corresponding tools provide little simulation and verification support. Flexibility also comes at the price of considerably more opportunity for errors.

The system-level languages SystemC and SpecC attempt to maintain the expressiveness of implementation languages, while enforcing some higher-level semantics, e. g. certain communication conventions between tasks (so-called "transaction-level modeling"). This favors structured design and reduces errors.

2.2 Target Architectures

In this thesis, we concern ourselves with system-level performance analysis, and therefore assume a system-level view of an embedded architecture. Consequently, architecture models below the system-level, including detailed models of processors, dedicated hardware blocks, memories etc., as well as instruction sets, gates, transistors, and physical models are not considered. On the other hand, our definition of 'system-level' goes beyond the typical hardware-designer's definition to include operating systems and communication protocols, which are required in complex embedded systems.

In our system-level view, an embedded architecture consists of computation resources and communication resources. *Computation* resources include (often specialized) programmable processors such as micro-controllers (μCs) or digital signal processors (DSPs), and dedicated or weakly-configurable co-processors (ASICs respectively FPGAs). A processor often executes an embedded real-time operating system (RTOS), e. g. VxWorks [119] in telecommunication applications, or OSEK-based RTOSes [66] such as ERCOSEK [29] in automotive applications, to schedule different software tasks, manage task access to local memory, control timers and react to external interrupts.

Communication resources include busses and networks, bus interface units (BIUs), and shared memory. Busses typically use standardized architectures, such as AMBA [3] or SiliconBackplane [104] for System-on-Chip (SoC) communication, and CAN [93] or the upcoming FlexRay [33] for communication between distributed automotive control units. Some bus architecture specifications include a protocol for bus arbitration (e. g. a static-priority collision-avoidance protocol in CAN), others support several arbitration mechanisms (e. g. a combination of TDMA and static priority scheduling in FlexRay), and yet others do not specify arbitration mechanisms (e. g. AMBA).

2.2.1 Systems-on-Chip

We are particularly interested in complex architectures consisting of many different components. In domains such as telecommunication or multimedia, different components are often integrated on a single chip, leading to so called systems on chip (SoC). Recently, the term 'multi-processor systems on chip' (MPSoC) has emerged to describe complex SoCs with multiple programmable cores and a complex communication structure. An SoC usually comprises different intellectual property (IP)-components from different vendors [28, 56], since a system integrator usually tries to re-use as many proven components as possible to concentrate on his added value. Typical IP components are the aforementioned types of processor cores with RTOSes, the aforementioned bus architectures, but also standard interface blocks and hardware-components dedicated to a particular computation function. Often, a complete domain-specific SoC *platform* is provided by a platform vendor (e. g. for mobile communication), which already consists of suitable components to perform many standard domain functions, and can be extended with special functions by an OEM (original equipment manufacturer, meaning a company who produces an end product). A platform also includes software and reference designs, thus further reducing the effort for an OEM to build a product. An example are the Philips Nexperia Home (Fig. 2.4)

and Nexperia Mobile platforms for multimedia and mobile digital audio applications [88].

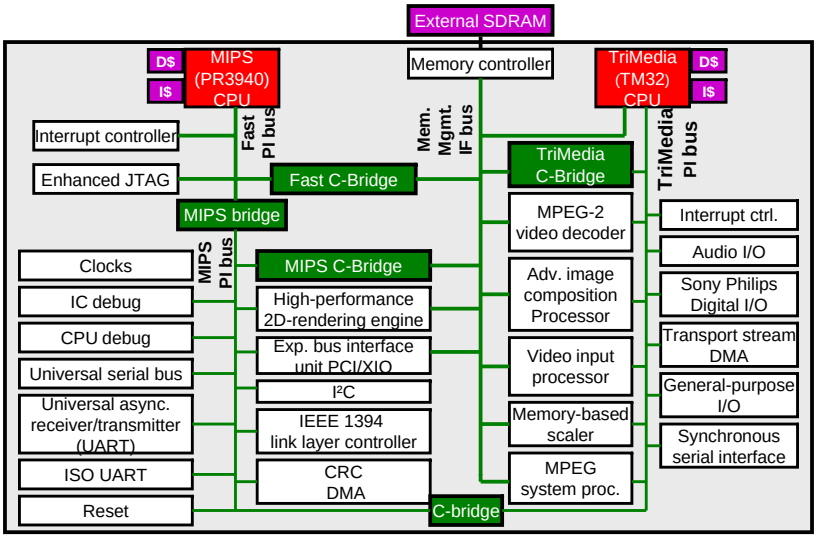


Figure 2.4. The Philips Nexperia Home platform for multimedia applications

2.2.2 Distributed Systems

In the automotive domain, a multitude (close to 100 in high-end vehicles) of dedicated electronic control units (ECUs) located in different parts of a vehicle are networked via the aforementioned CAN, as well as LIN [67], MOST [80] and (soon to come) FlexRay busses. Application-specific ECUs that already include specialized peripheral interfaces as well as so-called basic software are provided by ECU vendors. For example, an engine control ECU by default includes enough basic software that it is able operate an engine, albeit maybe not very efficiently [11]. All that remains for the automotive OEM is to calibrate the control algorithms on the ECU for optimal behavior, and potentially to add functions that distinguish him from competing OEMs [115].

A more traditional alternative is board-level integration, where several closely spaced boards with dedicated processors, memories and peripherals are connected via a backplane. An example is board-level integration based on the VME bus [43], which is often used in industrial control systems, aerospace and military applications.

2.3 Performance Constraints

Performance constraints are imposed by the environment in which an embedded system will operate. They constrain valid implementations of the system function. Performance constraints can be categorized as either *hard* or *soft*. A system implementation has to meet all hard performance constraints; otherwise the system does not function correctly. Hard performance constraints are typically timing constraints, e. g. deadlines, maximum allowed jitter, or minimum execution rate. An additional hard performance constraint is the avoidance of communication buffer over- or underruns that can corrupt system function.

Soft constraints are similar to optimization goals. Violation of soft constraints gradually reduces the usability, marketability, profitability etc. of the system, but does not lead to immediate failure of system functions.

2.4 Function Implementation and Integration

During function implementation, a function available in an implementation or system-level language is mapped onto a specific architecture component, using compilation (software) and synthesis (hardware), or manual re-code. As a result, an image of the function specification is obtained which is executable on the target architecture component.

If the architecture component is a programmable processor, then a user-function will interact with low-level software functions, in particular the operating system and drivers, which in turn use low-level hardware functions such as I/O-units, timers, interrupts control etc. Fig. 2.5 shows a typical hardware/software architecture on an embedded processor, with user-written application software resting on several levels of system hardware and software.

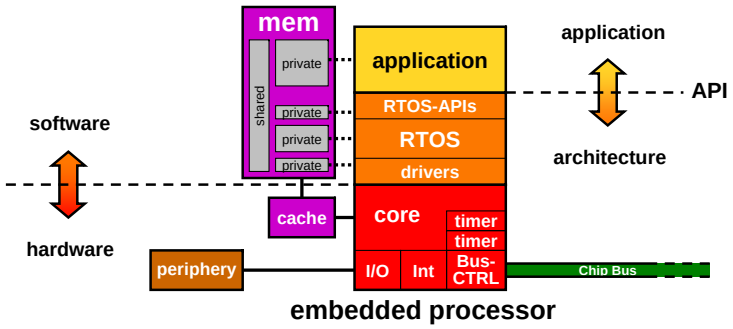


Figure 2.5. Typical hardware/software architecture on an embedded processor

Often, function structure is changed during implementation. Changes include the combination or splitting of functions, which may be implemented on different resources, and the change of communication mechanisms, e. g. from message passing to shared memory or vice versa.

The target architecture has a constrained number of computation and communication resources operating at finite speeds, as well as limited memories and communication buffers. Consequently, computation resources have to be shared between functions, and communication resources have to be shared between logical communication channels.

2.4.1 Task Activation and Scheduling

A task is activated due to an activating event. Activating events can be generated in a multitude of ways, including expiration of a timer, external or internal interrupt, and task chaining.

The purpose of a scheduler is to arbitrate between multiple tasks that want to simultaneously use a resource. A scheduler selects a task to which it grants the resource out of the set of active tasks according to some scheduling policy [13]. Other active tasks have to wait. The most complex schedulers are part of operating systems that schedule software tasks on embedded processors. But schedulers are also implemented in hardware to control access to other types of resources. For example, communication via a shared bus requires bus access scheduling, which is often performed by a bus-master implemented in hardware. Memory bandwidth can be considerably increased for certain types of memories, e. g. SDRAM, if memory-accesses are scheduled smartly [39]. A node in a network-on-chip needs to schedule different communication requests [17]. Three major classes of scheduling strategies can be distinguished.

1 Non-preemptive static execution order scheduling, mainly for highly regular digital signal processing (DSP) applications (section 2.1.1).

2 Priority-driven scheduling, often used in highly reactive systems operating in dynamic environments. Here, we further distinguish scheduling with fixed priorities and scheduling with dynamic priorities. In the fixed-priority case, priority assignments often follow a *rate-monotonic scheduling* (RMS) [68] or *deadline-monotonic scheduling* (DMS) [4] strategy, but priorities can be statically assigned in any other way. In the dynamic-priority case, priority assignments often follow an *earliest deadline first* (EDF) [68] strategy, but priorities can be dynamically assigned in any other way.

3 Preemptive time-slicing techniques, often used for a fair distribution of the resource among all tasks. Here, we further distinguish

static *time-division media access* (TDMA) scheduling with fixed-length time slots, and more dynamic *round-robin* (RR) scheduling with variable-length time slots [15].

A major advantage of static execution order scheduling is that it requires no run-time scheduler and hence incurs no *scheduling overhead*. However, the flexibility that is needed in modern embedded systems, as well as data-dependent task execution times and increased interaction with the system environment more and more necessitate dynamic scheduling. Static scheduling is thus typically limited to highly regular 'islands' [123] in an otherwise dynamic system. An example is the static ordering of tasks with the same period, but preemptive scheduling of tasks with different periods in the automotive operating system ERCOSEK [29].

TDMA scheduling essentially separates a real resource into multiple independent virtual resources, each with a guaranteed fraction of the available processing time or communication bandwidth. This is considered attractive for safety-critical applications in domains such as aerospace or automotive, in particular because prediction of response times is easy, and because faults in one virtual subsystem cannot impact a different virtual subsystem. However, TDMA scheduling is inefficient in the presence of dynamically changing loads, since a time-slice large enough to handle a worst-case load remains partially unused in all other cases. TDMA is also poorly suited to achieve short response times. Unless a system is globally synchronized, which is problematic (section 2.1.2), an activation may wait for a full turn (the sequence of all time slots) in the worst-case until it is granted a TDMA resource. It will also be interrupted for additional turns if it does not complete within one time slot. RR scheduling resolves the problem of underused TDMA resources, but short response times still cannot be achieved. Additionally, the calculation of worst- and best-case response times is complex. RR scheduling is rarely used in hard real-time applications.

Reactive, priority-based scheduling does not have the inefficiencies of TDMA. In particular, it is well suited for multi-processor systems, where a task is activated by the arrival of data from a different task, often mapped onto a different component. In multi-processor systems, arrival timing of data often jitters or can even arrive in bursts. High-priority tasks can immediately react to data arrival, thus ensuring short response times. The tasks also do not waste resources if they are idle, instead leaving a resource fully to lower-priority tasks. While EDF-scheduling has been proven to be theoretically optimal in the sense that if EDF cannot guarantee that all deadlines are met, no other scheduling

strategy can [68], it produces considerable scheduler overhead, since priorities have to be frequently re-calculated. Static priority scheduling is theoretically less efficient, but scheduler implementation is simple, and the smaller scheduler overhead often compensates for the theoretical inefficiencies [14].

2.5 Integration Problems

The long list of differences between executable specification and implementation imply that the architecture idealizations assumed during function specification are no longer valid. This leads to a variety of integration problems which generally fall into three categories: *functional problems*, *resource problems* and *performance problems*.

Typical functional problems during integration include different interpretations of ambiguous functional requirements by different design-teams, mismatches in bit-width, endian-ness or physical unit, and improper handling of special situations, e. g. system startup. Typical resource problems during integration include insufficient memory to hold all code and data at the same time, or deadlock due to faulty resource arbitration. Functional problems can be debugged quite well using the existing test-bench (though it is rarely complete and consequently some bugs remain undetected). Code memory problems are usually detected at linkage time. Data memory problems can be hard to detect in programs that allocate memory dynamically (either on the heap or on the stack). Tools like Purify [94] help. Deadlocks can be avoided by proper resource-allocation protocols, e. g. the priority ceiling or priority inheritance protocols [102]. These problems are not further considered in this thesis.

Performance problems typically arise due to finite resource speed, and in particular due to resource sharing as a result of function- and subsystem-integration. They include the violation of deadlines, failure to sustain a required execution rate, buffer over- or under-run, or output jitter beyond the acceptable limit. For example, in Fig. 2.6 tasks belonging to one subsystem are first integrated onto shared processing components (shown for subsystem 1). Then, functionally independent subsystems are integrated onto a common architecture. In that process, the double-arrowed communication links in both subsystems are mapped onto a shared bus. Both integration steps create performance dependencies between tasks. In reality, the number of performance dependencies is much larger, since processors are also shared between subsystems, and communication also occurs across subsystems boundaries. Tracking performance dependencies can be extremely difficult, in particular

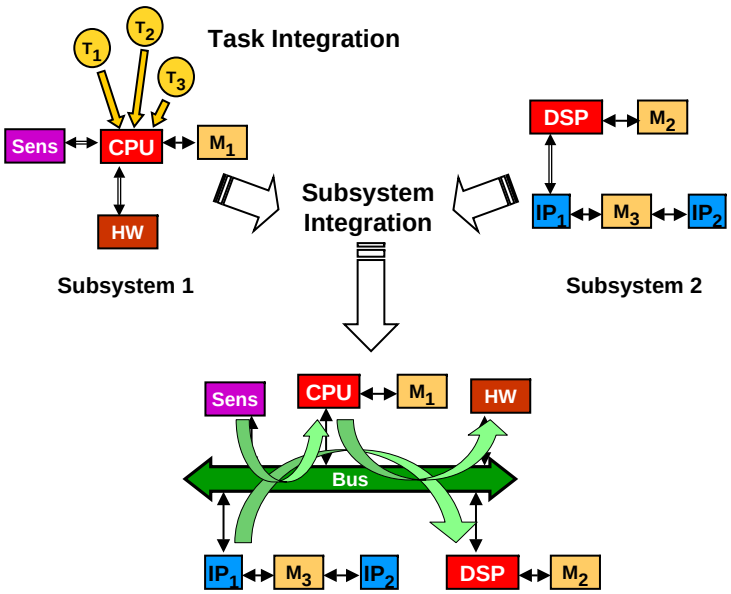


Figure 2.6. Performance dependencies between functionally independent subsystems introduced by resource sharing

if neither the system integrator nor the subsystem designers understand the complete system.

State-of-the art to debug performance problems are test and cycle-accurate system-simulation. Test requires the (prototyping) hardware to be fully available, which limits test to the last design stage. Errors found at this stage often lead to expensive re-designs. Cycle-accurate system-simulation with tools like Axys MaxSim [5] or Mentor Graphics Seamless [78] alleviates this problems by allowing to find problems before hardware has been produced, and allowing reasonably fast and inexpensive architecture changes. Cycle-accurate system-simulation additionally offers superior debugging support, since internal state can be visualized beyond what is accessible on the actual hardware, and since debugging does not interfere with the system. Still, cycle-accurate system-simulation requires detailed architecture models and fully implemented functions, and is thus also confined to late design stages.

2.6 The SPI Project

The goal of the SPI project is to enable system-wide analysis of non-functional system properties, in order to allow safe and reliable integration and optimized implementation of heterogeneously specified multi-

language embedded systems [126, 123, 44]. The acronym SPI is derived from (**S**ystem **P**roperty **I**ntervals).

The key concepts behind SPI are *abstraction* of functional and architecture details and the use of *intervals*. Abstraction allows to focus entirely on non-functional system properties, in particular those that influence performance. The use of intervals leads to a single description of many different behaviors in the presence of uncertainty. Together, abstraction and the use of intervals lead to system-representations which are excellent for performance analysis and design-space exploration.

Abstraction and the use of intervals have several further advantages. A major advantage is the ability to effectively cope with complex embedded systems [123], such as those described in this chapter. Furthermore, a design-flow with incomplete or partially hidden function and architecture specifications is naturally supported. As long as properties of function or architecture components that influence performance are available, performance analysis is possible without further knowledge of internal details. Incomplete specifications are the rule in early design-stages, while hidden internal details are the result of component integration from different design teams.

2.6.0.1 The SPI Model

The SPI project initially focused on capturing non-functional system properties from a variety of MOCs, in order to be independent of specific MOCs, and in particular to support multi-language design. The idea was to transform a heterogeneous function specification into a homogeneous, *internal* representation of non-functional system properties, which would then be used as a basis for *performance analysis* and *coordination synthesis* [28], i. e. the control of mapping and scheduling decisions as well as the synthesis of function interfaces during function integration. The goal was to avoid many of the integration problems that today are discovered late in the design when function implementations are finally integrated and co-simulated.

Ziegenbein developed the SPI model [126, 123] as an internal representation for this purpose. SPI captures communication and coordination as a process network, but abstracts detailed process behavior. The SPI model is sufficiently expressive to capture a large variety of different communication and coordination semantics common in different MOCs, and thus can in fact serve as a unifying representation that allows to consider all application components during performance analysis independent of the modeling languages or tools used. To prove this claim, model transformations from Simulink to SPI [48, 47], as well as from

SDL to SPI [53] have been developed. The transformation from several other MOCs to SPI was considered in [95].

In SPI, *processes* communicate via two different channel types, *FIFO-queues* and *registers*. The activation of SPI processes is implicit and based on data availability, i. e. a process *may* start if there is sufficient data on its input queues to support one execution. A process is modeled by an activation function, an execution time, as well as consumer and producer data rates at task inputs. Data is produced on and consumed from all connected communication channels simultaneously at the end of the execution of a process (*atomic buffer update*). Channels can be initialized with tokens.

Process and channel properties are specified as *value intervals*. For example, a data rate interval limits the number of produced or consumed data, capturing conditional communication. Using the concept of *process modes*, such conditional process behavior depending on internal states or input data can also be modeled explicitly, i. e. a process is refined to have different behaviors (parameter sets) that are modeled as modes. *Virtual* processes and channels are used to model the system environment and to represent scheduling constraints.

Finally, *timing constraints* can be specified in SPI using latency path constraints that limit the time for causal process executions along a certain path. Other types of timing constraints (rate, jitter etc.) can be modeled by latency path constraints over virtual elements. A formal description of the SPI model can be found in [126, 123].

2.6.0.2 The SPI Workbench

The SPI model by itself is generally not well suited for performance analysis, since the modeling of preemptive resource sharing, as well as the modeling of timing in event streams are awkward. The modeling of resource sharing effectively requires describing state machines that model the scheduler and the different scheduling states that each process can assume. This is both complex and unwanted, since the benefit of the SPI model, a pure representation of application coordination and communication, would be completely blurred by a multitude of implementation-dependent components. The same hold for the modeling of possible timing in event streams using SPI. For example, Fig. 2.7 shows a SPI representation of an event source which produces an output event on average every 4 time units. The period is not strictly maintained. Instead, each event can be delayed from its perfect position by up to 1 time unit. The period is enforced by the latency constraint $LC = 4$ on channel *cp*. The delay interval stems from the latency constraint interval $LC = [0, 1]$ on channel *cj*. The possible timing at the output of *cj* would have to be

detected by a sophisticated analysis algorithm. In the next chapter we will see a much more efficient way to express timing information.

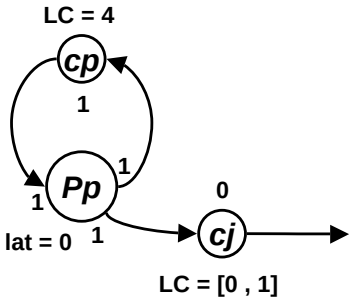


Figure 2.7. SPI representation of an event source which produces an output event on average every 4. Each event can be delayed from its perfect position by up to 1 time unit

In conclusion, SPI combines several key concepts which make it a very efficient model to describe *application coordination*, which is fundamental for formal system-level performance analysis. In particular, these concepts include abstraction of functional details and the use of intervals. The choice between FIFO and register communication, together with process modes and mode tags to capture correlation between modes, as well a virtual elements allow to efficiently capture performance-relevant data at various levels of detail.

However, SPI cannot capture efficiently neither the target architecture, nor scheduling effects. This is obvious when comparing timing and scheduling modeled directly in SPI [124] against an approach where an additional representation better suited for scheduling analysis is used [127].

2.7 Summary and Conclusion

In this chapter we gave an overview over system-level design. We emphasized the complexity and heterogeneity of both embedded applications and embedded architectures. We identified system-level performance verification as a key design challenge, stemming from the huge number of performance-interdependencies that arise from resource-sharing and scheduling due to function- and subsystem-integration. We emphasized the need to consider performance as early as possible in the design flow, due to the large cost of fixing performance problems discovered late in the design. Considering performance early also allows design-space exploration, leading to optimized solutions which can make the difference between a successful product and one that cannot be sold profitably.

We argued that cycle-accurate simulation, which is state-of-the-art in performance verification, is poorly suited both for early design stages and for design-space exploration, since it requires detailed architecture models and fully implemented functions. We then hinted at an alternative, formal performance analysis. The SPI-project took a first step in that direction by systematically capturing application properties into a representation that abstracts functional details and focuses exclusively on performance-relevant properties (separation of concerns). In particular, SPI captures a large variety of activation dependencies which occur in embedded applications (application coordination model).

Our overview of MOCs and languages showed that activation dependencies common in embedded applications are AND- and OR-activation, cyclic dependencies, conditional communication, data-rate transitions and data-dependent behavior. Any useful system-level performance analysis framework must be able to deal with these complex application dependencies. In the next chapter we will see that representations for scheduling analysis also use abstraction and intervals to model the possible scheduling dependencies (i. e. *execution coordination*) between tasks. On the other hand, the application models in these representations are a far cry from the expressiveness of common MOCs and languages. The focus of this thesis will thus be to combine powerful scheduling representations with powerful application representations, in order to enable performance analysis of real-world systems.

Chapter 3

PERFORMANCE ANALYSIS

System-level performance analysis calculates conservative bounds for system performance. The most widely researched approaches use *scheduling analysis*, which constructs worst-case (and sometimes best-case) scenarios, and calculates the performance of these scenarios.

In this chapter, we present several system-level performance analysis approaches based on scheduling analysis. Our main focus is a novel compositional performance analysis approach developed by our group. We then point out that existing performance analysis techniques in general do not support the complexity of real-world embedded applications. We also give a brief overview over alternative performance analysis approaches that are not based on scheduling analysis.

3.1 Scheduling Analysis

When multiple computation (communication) tasks share one computation (communication) resource, then two or more tasks may request the resource at the same time. Scheduling resolves these conflicting requests (section 2.4.1). To properly design an embedded real-time systems it is important to understand the effect that scheduling has on system performance.

Scheduling analysis calculates worst-case (sometimes also best-case) task response times, i. e. the time between task activation and task completion, for all tasks sharing a resource under the control of a scheduler. Scheduling analysis guarantees that all observable response times will fall into the calculated [best-case, worst-case] interval. We therefore say that scheduling analysis is *conservative*.

Worst-case response times can then be compared against deadlines. If all deadlines are met in the worst case, then they will be met in all cases,

thus guaranteeing that a system will satisfy all deadline constraints. Obviously, the calculated response times should also be as tight as possible to avoid over-dimensioning of a system.

Scheduling analysis requires information about certain system properties (section 3.2), and about the possible timing of task activations. The possible timing of task activations is efficiently described using *event models*.

3.1.1 Event Models

An event model describes the possible timing of events. It can be expressed using two *event functions* $\eta^u(\Delta t)$ and $\eta^l(\Delta t)$.

Definition 3.1 (Upper Event Function) *The* upper *event function* $\eta^u(\Delta t)$ *specifies the* maximum *number of events that can occur during any time interval of length* Δt.

Definition 3.2 (Lower Event Function) *The* lower *event function* $\eta^l(\Delta t)$ *specifies the* minimum *number of events that have to occur during any time interval of length* Δt.

Event functions are piecewise constant step functions with unit-height steps, each step corresponding to the occurrence of one event. An example is shown in Fig. 3.1. Note that at the points where the functions step the smaller value is valid for the upper event function, while the larger value is valid for the lower event function (indicated by dark dots). We will explain this example in more detail in a minute. For any time interval of length Δt, the actual number of events is bound by the upper and lower event functions. Event functions resemble arrival curves [23, 87] which have been successfully used by Thiele et al. for compositional performance analysis of network processors [109]. We will take a closer look at their work in section 3.7. In the following, the dependency of η^u and η^l on Δt is omitted for brevity.

Event models can also be described by sets of parameters, together with rules how to construct both event functions from them. For example, a *periodic with jitter* event model has two parameters ($\mathcal{P}$, $\mathcal{J}$) and states that each event generally occurs periodically with period $\mathcal{P}$, but that it can jitter around its exact position within a jitter interval $\mathcal{J}$. A *periodic with jitter* event model is described by the following event functions $\eta^u_{\mathcal{P}+\mathcal{J}}$ and $\eta^l_{\mathcal{P}+\mathcal{J}}$ [98].

$$\eta^u_{\mathcal{P}+\mathcal{J}} = \left\lceil \frac{\Delta t + \mathcal{J}}{\mathcal{P}} \right\rceil \tag{3.1}$$

$$\eta^l_{\mathcal{P}+\mathcal{J}} = \max\left(0, \left\lfloor \frac{\Delta t - \mathcal{J}}{\mathcal{P}} \right\rfloor\right) \tag{3.2}$$

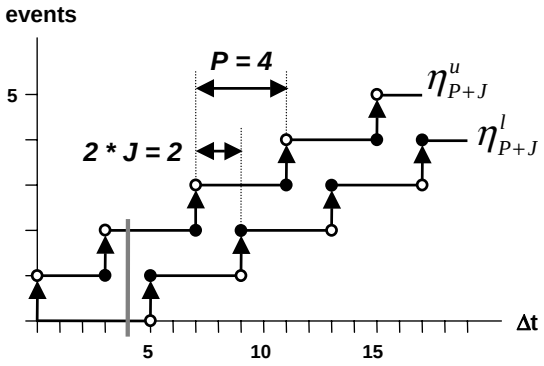


Figure 3.1. Upper and lower event functions for a *periodic with jitter* event model with ($\mathcal{P} = 4$, $\mathcal{J} = 1$)

Consider an example where

$$(\mathcal{P}, \mathcal{J}) \quad = \quad (4, 1)$$

This event model is visualized in Fig. 3.2. Each gray box indicates a jitter interval of length $\mathcal{J} = 1$. The jitter intervals repeat with the event model period $\mathcal{P} = 4$. The figure additionally shows a sequence of events which satisfies the event model, since exactly one event falls within each jitter interval box, and no events occur outside the boxes. An infinitely long sequence of events is called an *event stream*. If all events in the event stream satisfy an event model, then the event stream is said to satisfy the event model.

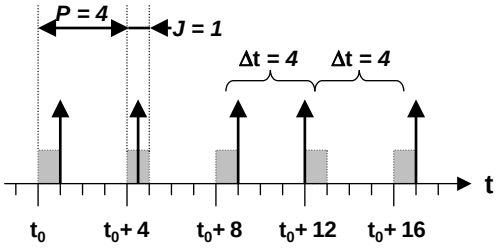


Figure 3.2. Example of an event stream that satisfies a *periodic with jitter* event model with ($\mathcal{P} = 4$, $\mathcal{J} = 1$)

Let us now return to the event functions in Fig. 3.1. These event functions correspond to our *periodic with jitter* event model in accordance with equations 3.1 and 3.2. To get a better feeling for event functions, imagine a sliding window of length Δt that is moved over the (infinite) length of an event stream. Consider $\Delta t = 4$ (gray vertical bar in

Fig. 3.1). The upper event function indicates that at most 2 events can be observed during any time interval of length $\Delta t = 4$. This corresponds e. g. to a window position between $t_0 + 8.5$ and $t_0 + 12.5$ in Fig. 3.2. The lower event function indicates that no events have to be observed during $\Delta t = 4$. This corresponds e. g. to a window position between $t_0 + 12.5$ and $t_0 + 16.5$ in Fig. 3.2.

Let us further introduce *distance* functions $\delta^{min}(N \geq 2)$ and $\delta^{max}(N \geq 2)$, which return the minimum respectively maximum distance between $N \geq 2$ consecutive events in an event stream.

Definition 3.3 (Minimum Distance Function) *The* minimum *distance function* $\delta^{min}(N \geq 2)$ *specifies the* minimum *distance between* $N \geq 2$ *consecutive events in an event stream.*

Definition 3.4 (Maximum Distance Function) *The* maximum *distance function* $\delta^{max}(N \geq 2)$ *specifies the* maximum *distance between* $N \geq 2$ *consecutive events in an event stream.*

For *periodic with jitter* event models we obtain

$$\delta^{min}(N \geq 2) = \max\{0\,,\ (N-1) * \mathcal{P} - \mathcal{J}\} \tag{3.3}$$

$$\delta^{max}(N \geq 2) = (N-1) * \mathcal{P} + \mathcal{J} \tag{3.4}$$

For example, the minimum distance between 2 events in a *periodic with jitter* event model with ($\mathcal{P} = 4$, $\mathcal{J} = 1$) is 3 time units, and the maximum distance between 2 events is 5 time units.

Periodic with jitter event models are well suited to describe generally periodic event streams, which often occur in control, communication and multimedia systems [97]. If the jitter is zero, then the event model is strictly periodic. If the jitter is larger than the period, then two or more events can occur at the same time, leading to bursts. In the presence of bursts, a *periodic with jitter* event model can be extended with a d_{min} parameter that captures the minimum distance between events within a burst. A more detailed discussion can be found in [98].

Additionally, *sporadic* events are also common [97]. We model sporadic event streams with the same set of parameters as periodic event streams. The difference is that for sporadic event streams, the lower event function $\eta^l(\Delta t)$ is always zero. The maximum distance function $\delta^{max}(N \geq 2)$ approaches infinity for all values of N [98]. Note that *jitter* and d_{min} parameters are also meaningful in sporadic event models, since they allows to accurately capture sporadic transient load peaks.

Other event model representations have been proposed. Gresser [35, 36] proposes a very general formalism that uses tuples which capture

the maximum number of events that can occur for certain Δt. It is possible to express our event models using Gresser's notation. However, system-level performance analysis has been demonstrated only for earliest deadline first (EDF) scheduling where deadlines were locally assigned and true completion times were not taken into account. Therefore, each resource could be analyzed separately, which as we will see below is not possible in general.

In section 2.6 we showed that event models can also be represented directly in SPI. In fact the example we used there represents the same *periodic with jitter* event model that we use in this section. As can be seen, the modeling effort is considerably larger in SPI. The SPI representation of event models resembles timed automata which are used for model checking-based scheduling analysis (see below).

3.2 Single-Processor Scheduling Analysis

Single-processor scheduling analysis is a local analysis technique applicable to one scheduling component (typically a processor, hence the name, but the same approach can be used e. g. to analyze bus-arbitration). A classic example is *rate-monotonic analysis* (RMA) by Liu & Layland [68], which analyzes if tasks scheduled according to the *rate-monotonic scheduling* (RMS) policy can meet their deadlines. This work was generalized by Joseph and Pandya to static-priority preemptive scheduling on a single processor [54]. The following equation gives the worst-case response time r_i of a lower priority task i due to a worst-case number of interrupts by all higher priority task $j \in hp(i)$. C_i, C_j are the worst-case core execution times (WCET, i. e. assuming no interrupts) of tasks i and j. The equation holds only if r_i is smaller than the minimum distance between two activations of task i, and ignores context-switch overhead.

$$r_i \quad = \quad C_i + \sum_{\forall j \in hp(i)} \eta_j^u(r_i) * C_j \tag{3.5}$$

The term r_i appears on both sides of this equation, which thus cannot be solved directly. Audsley presented an iterative algorithm to obtain the desired result. It is first assumed that $r_i = C_i$. It is then checked, for how long task i can be interrupted during this time. The sum of all possible interrupts is added to r_i, and the check is repeated, until a fixpoint is reached (success) or r_i exceeds the minimum distance between two activations of task i (abort).

Worst-case and best-case core execution times can be obtained in a variety of ways. A task can be executed in isolation on the target architecture for a range of stimuli in search of worst-case and best-case paths.

This is much easier for a single task than for a whole system, because the designer of the task knows the code very well. A more reliable approach is static execution-time analysis [120], which considers all possible stimuli to calculate conservative worst- and best-case core execution times. Core execution times can also be guessed in early design-stages before a task has been coded, e. g. based on values from previous designs.

Single-processor scheduling analysis has improved over time to reduce constraints under which the analysis works, e. g. consideration of arbitrary deadlines [65] or more complex activation timing [105, 35], and to support different scheduling techniques [35, 46]. Additionally, scheduling effects have been modeled in more and more detail, including blocking-times [102], RTOS-overhead [114] and various types of correlations between different task activations [113, 79, 85, 7]. A good overview of single-processor scheduling analysis techniques can be found in [32, 13, 96]. Single-processor scheduling analysis has also found its way into industrial practice. TriPacific's Rapid RMA [116], LiveDevices' Real-Time Architect (RTA) [69], and TimeWiz by TimeSys [111] all calculate worst-case task response times for single processors with static priority scheduling. RTA models the scheduling influences of a specific automotive RTOS in detail.

We have also used a real-world RTOS from the automotive domain to show that scheduling analysis accuracy can be increased to a point where the calculated worst-case response time is only a few percent longer than the longest measured response-time [52]. High accuracy comes at the cost of higher model and analysis complexity, and consequently higher modeling and analysis times. However, even at a high level of detail, scheduling analysis run-time compares favorably to the time required for cycle-true simulation [50].

One major open issue in single-processor scheduling analysis are caches. In the past, embedded processors rarely had caches, or they were turned off for hard real-time applications, because predictability of cache-behavior was poor. Recently, performance requirements for embedded processors and the size of embedded applications have grown so much that caches are becoming commonplace. Examples for embedded processors with caches include PowerPC, StrongARM, or TriCore. Extension of scheduling analysis to consider cache-effect has become an interesting field of research [59, 81].

3.3 Homogeneous Multi-Processor Scheduling Analysis

Single-processor scheduling analysis cannot handle multi-processor architectures, which are common in modern embedded systems. The sim-

plest form of multi-processor scheduling analysis are extensions to the single-component analysis techniques described above, e. g. [101]. Communication between tasks is via shared memory. Multiprocessor analysis applies the component-level techniques locally, and captures task dependencies by creating relations between certain process timing parameters and the response time equations. It is restricted to homogeneous scheduling for all tasks.

As was shown by Yen and Wolf [122], in multi-processor scheduling analysis it is generally not sufficient to consider only worst-case response times. For example, a shorter response time of task a on one resource can lead to an earlier activation of a dependent task b on a different resource, which my then preempt a second task c on that resources that would not have been preempted if task a had had a longer response time. This may cause task c to miss its deadline. Such *scheduling anomalies* can only be reliably detected if both worst- and *best*-case response times are calculated during scheduling analysis.

3.4 Holistic Scheduling Analysis

"Holistic" in the context of scheduling analysis refers to a consistent end-to-end response time analysis approach for multi-processor real-time systems, where processors communicate over a bus. The maximum response time of a task is treated as the release jitter for the next task along a task dependency chain. The first such approach by Tindell considers processes scheduled under RMS which exchange messages via a TDMA bus [112]. Newer approaches by Eles et al. consider a larger variety of mixed time/event-triggered distributed embedded systems, and focus on real-world communication protocols [90–92]. They consider the *time-triggered protocol* (TTP) [57] as a specific TDMA implementation, and *control area network* (CAN) [93] as a specific static-priority implementation. Garcia and Harbour refined Tindell's basic approach in several ways to improved analysis tightness, including static and dynamic offsets, arbitrary deadlines [85] and best-case analysis techniques to improve worst-case analysis in the presence of scheduling anomalies [84].

Flexibility, subsystem integration and scalability are major weaknesses of holistic techniques. If an architecture changes, then it is easily possible that a holistic analysis does no longer fit the problem. This means that an architecture cannot be composed as desired without compromising analyzability. The problem is aggravated if subsystems, in particular IP components need to be integrated. Then, a local performance model for a subsystem may not satisfy holistic assumptions. For example, holistic analyses by Tindell, Garcia and Harbour assume periodic events at system inputs. This means that a subsystem cannot be

analyzed, if it is connected to a second subsystem with output jitter. End-to-end delay calculation across subsystems is also not supported.

3.5 Compositional Performance Analysis

Compositional performance analysis [97, 98, 109] enables performance analysis for complex, heterogeneous embedded architectures and supports subsystem integration, thus overcoming the restrictions of other techniques. It works by integrating different local scheduling analysis techniques into a system-level analysis. This process mimics the integration of different architecture components into the system architecture. The local scheduling analysis techniques are composed on the system level by connecting their input and output event streams according to the overall application and communication structure.

Obviously, for compositional performance analysis to work, local scheduling analysis techniques must be available for each architecture component. This is not necessarily the case. However, each newly programmed local scheduling analysis can be added into a library of analysis techniques for re-use in future systems. This reduces the amount of analysis programming to novel components that have not been considered before. The effort to scale to larger and more complex heterogeneous systems is thus minimized. This is a major advantage over 'holistic' multi-processor analysis approaches [114, 90]. On the other hand, holistic approaches may make it easier to take global performance effects into account than compositional performance analysis, potentially yielding tighter analysis bounds.

In this section, we mainly focus on a novel compositional performance analysis approach developed by Richter et al. [97, 98]. The performance analysis methodology for complex embedded applications, which is the subject of this thesis, builds upon Richter's work. We additionally present a compositional performance analysis methodology developed by Thiele et al. [109], which uses a novel local analysis approach and is geared towards the analysis of network processors. Richter's approach has the advantage that it allows to re-use the large number of existing analysis techniques developed by the real-time community, such as those introduced in section 3.1. It is well suited for several important embedded systems domains, including telecommunication, multi-media and control systems. To the best of our knowledge, Richter's and Thiele's approach are the only two compositional performance analysis approaches currently available for heterogeneous embedded architectures.

3.6 Composition Using Standard Event Models

In the compositional performance analysis methodology developed by Richter et al. [125, 97, 98] (henceforth called 'our methodology'), input and output event streams are described by event models with *period*, *jitter*, and *minimum distance* parameters. Additionally, *periodic* and *sporadic* event streams are distinguished. These event models were introduced in detail in section 3.1.1.

Event models with this small set of parameters have several advantages. Firstly, they are easily understood by a designer, since period, jitter etc. are familiar event stream properties. Secondly, the corresponding event functions and distance functions can be evaluated quickly, which is important for scheduling analysis to run fast. Thirdly, as we will shortly see, compositional performance analysis requires the modeling of possible timing of output events for propagation to the next scheduling component. Our event models allow us to specify simple rules to obtain output event models that can be described with the same set of parameters as the activating event models. Therefore, we do not have to depart from our event models independent of size and structure of the composed system (hence the term 'standard'). This makes our compositional performance analysis approach very general.

Throughout this thesis, slightly simpler *periodic with jitter* and *sporadic with jitter* event models with parameters $(\mathcal{P}, \mathcal{J})$ but without a minimum distance parameter d_{min} will be used. This is sufficient to illustrate our ideas for compositional performance analysis for complex applications. It should be emphasized that this is not a restriction of the work presented in this thesis. It can be extended to also consider d_{min}, and is equally valid for any other type of event model representations possibly used in compositional performance analysis.

3.6.1 Task Model

Our existing approach assumes that each task has one input FIFO. A task reads its activating data from its input FIFO and writes data into the input FIFO of a dependent task. A task may read its input data at any time during one execution. We therefore assume that the data needs to be available at the input during the whole execution of the task. When calculating the required size of input FIFOs we assume that input data is removed from the input FIFO at the end of one execution.

We also assume that a task writes its output data at the end of one execution. This assumption is standard in scheduling analysis. It is required to deduce the timing of output data from the calculated task response times. Tasks that deviate from this model can be split into a

chain of sub-tasks, each of which writes data at the end of its execution [85].

3.6.2 Output Event Model Calculation

An activating event model describes the possible timing of task activations on a resource. As was explained in section 3.1, this is required for response time calculation during scheduling analysis. An important aspect of our approach is that we not only calculate maximum response times, but also minimum response times. The importance of minimum response times to detect scheduling anomalies has been pointed out by several groups [122, 84] (see also section 3.1).

Our approach goes an essential step further and calculates the resulting output event model. An output event model describes the possible timing of task completions. The output event model period obviously equals the activation period. The difference between maximum and minimum response times (the response time jitter) is added to the activating event model jitter, yielding the output event model jitter[1].

$$\mathcal{J}_{out} = \mathcal{J}_{act} + (t_{resp,max} - t_{resp,min})$$

For an intuitive explanation of this formula, recall that the event model jitter captures the size of an interval within which each event can deviate from its exact periodic position. Imagine that the nth activating event occurs as soon as possible, and is processed in the minimum response time. The $n+1$ st activating event occurs as late as possible, and is processed in the maximum response time. Then, the maximum distance between output events has increased by $t_{resp,max} - t_{resp,min}$ compared to the maximum distance between activating events. A similar argument can be made for the minimum distance between output events, thus explaining the jitter increase.

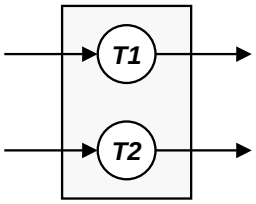


Figure 3.3. Example to illustrate output event model calculation

[1] In certain, a tighter output event model jitter can be calculated.

Consider the example in Fig. 3.3. Tasks $T1$ and $T2$ are mapped onto a common resource with static priority preemptive scheduling. The following task parameters are relevant for scheduling analysis: *priority*, *activation period*, *activation jitter*, and *core execution time* interval. Let us assume parameter values as given in the following table (a smaller number indicates a higher priority). The table also shows the calculated *response time* intervals, *output jitters* and, in case of $T2$, *minimum distance* between output events.

task	prio.	act. period	act. jitter	exe. time	resp. time	out. jitter	min. dist.
$T1$	1	6	1	$[2, 3]$	$[2, 3]$	2	–
$T2$	2	20	5	$[6, 9]$	$[8, 24]$	21	8

Table 3.1. Input and calculated parameter values for both tasks from Fig. 3.3

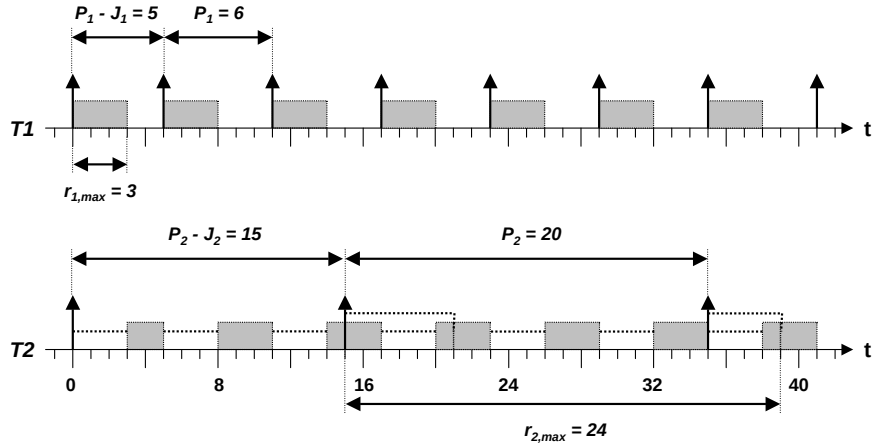


Figure 3.4. Scenario leading to the worst-case response times for both tasks from Fig. 3.3

Figs. 3.4 and 3.5 show the scenarios leading to the worst-case respectively best-case response time for both tasks. Note that in general the best-case calculation is NP-complete, but conservative polynomial approximations have been published [84]. Fig. 3.6 shows a scenario leading to the minimum and maximum distance between output events for task $T1$, from which we obtain the output jitter. The calculation for $T2$ works accordingly.

As can be seen, for both tasks the output jitter equals the activation jitter plus the difference between the maximum and minimum response

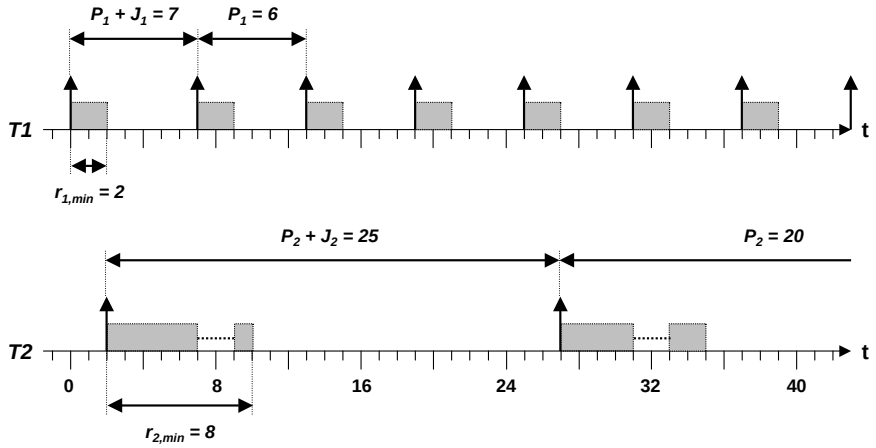


Figure 3.5. Scenario leading to the best-case response times for both tasks from Fig. 3.3

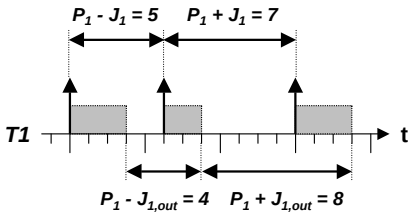


Figure 3.6. Scenario leading to the minimum and maximum distance between output events for task $T1$ from Fig. 3.3

times. The response time interval of $T1$ equals its execution time interval, since $T1$ cannot be preempted. I. e. for task $T1$, the output jitter increases only due to its non-constant execution time.

For task $T2$, the jitter increases due to a combination of its execution time interval, a non-constant number of preemptions by task $T1$, and a possible activation backlog (an activation of $T2$ occurs before the previous activation has been fully executed). In the best case $T2$ is preempted by $T1$ once, resulting in a shortest possible interrupt of 2 time units (the minimum execution time of $T1$). In this situation, no backlog is possible. Consequently, The minimum response time of task $T2$ is $6 + 2 = 8$ time units. In the worst case $T2$ is preempted by $T1$ several times, and activation backlog occurs, leading to a maximum response time of 24 time units.

Note, that the output event model of task $T2$ has a larger jitter than period. This information alone would indicate that an early output

event could occur before a late previous output event, which obviously cannot be correct. In reality, output events cannot follow closer than the minimum response time of a task. This is indicated by the value of the *minimum distance* parameter. For $T1$, the minimum distance of output events is implicitly its output period minus its output jitter.

Further effects, such as blocking times or scheduler overhead, have to be modeled for accurate response-time and output event model calculation. They are omitted here because they are not needed to understand the ideas presented in this thesis. Further details can be found in [96].

3.6.3 Analysis Composition

We assume that a task sends output data when it finishes execution. Tasks that do not satisfy this model can be split into a chain of subtasks for which the model is valid (section 3.6.1). The sending of output data at task completion time is interpreted as one output event in an event stream that satisfies the calculated output event model. Our compositional performance analysis allows to connect a task output with the input of a different task. Consequently, the output event model becomes the activating event model of the second task and can be used for scheduling analysis for that task.

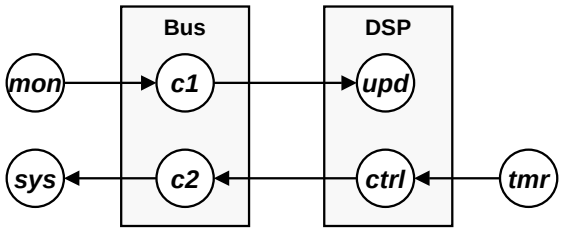


Figure 3.7. Example to illustrate compositional performance analysis

Consider the example in Fig. 3.7. Monitor task *mon* sends data that satisfies an arbitrary event model via communication task *c1* to update task *upd*. Control task *ctrl*, which is activated periodically by timer *tmr*, sends data via communication task *c2* to a system interface *sys*. Tasks *upd* and *ctrl* are both mapped to the same *DSP*, while both communication tasks share the same *Bus*. The idea of compositional performance analysis is to perform scheduling analysis of both resources separately, and to propagate output event models on the event streams between the two resources. Our compositional performance analysis approach makes no assumptions about scheduling policies and scheduling parameters and merely requires that scheduling analysis techniques are available for both

resources. The example allows us to study several interesting aspects of or compositional performance analysis approach.

3.6.4 Starting Point

Initially, only event models at the external system inputs are known. As explained in section 3.1, local scheduling analysis generally requires an activating event model for each task on a resource. Consequently, in our example in Fig. 3.7 scheduling analysis initially cannot be performed neither for the *Bus* nor for the *DSP*, since there is a cyclic scheduling dependency between the functionally independent task graphs. Scheduling analysis for the *Bus* requires an activating event model for task *c2*, which is obtained only after scheduling analysis on the *DSP*, which in turn requires an activating event model for task *upd*, which is obtained only after scheduling analysis on the *Bus*.

One solution to the starting point problem is to initially propagate all external event model periods and jitters along each task graph [96]. This approach is safe since on one hand scheduling cannot change an event model period. On the other hand, as will be explained shortly, scheduling can only *increase* an event model jitter [114]. Since a smaller jitter interval is contained in a larger jitter interval, the minimum initial jitter assumption is safe.

3.6.5 System-Level Analysis Iteration

In our compositional performance analysis approach we alternate local scheduling analysis and event model propagation [98]. A global analysis step consists of two phases [98]. In the first phase local scheduling analysis is performed for each resource. In the second phase, all output event models are propagated. It is then checked if the first phase has to be repeated because some activating event models are no longer up-to-date, meaning that a newly propagated output event model is different from the output event model that was propagated in the previous global analysis step. Analysis completes if either all event models are up-to-date after the propagation phase, or if an abort condition, e. g. the violation of a timing constraint has been reached.

Event model *periods* do not change during global analysis. Even model *jitters* can only increase. For an intuitive explanation, imagine that in Fig. 3.7 task *upd* interrupts task *ctrl*. The smaller the activation jitter of *upd* and *ctrl*, the tighter the bounds for the minimum and maximum number of interrupts of *ctrl* by *upd*, and hence the output jitter of *ctrl*. If any of the activation jitters increases, then the output jitter obviously cannot decrease, but it may increase. This argument

can be made for every resource in system. Therefore, propagation of an increased output jitter can only increase other output jitters, which are in turn propagated. A fix-point can be reached, since an activation jitter increase for one task does not increase output jitter of a different task if it does not change the minimum and maximum number of interrupts [98].

3.6.6 Event Stream Adaptation

A second key property of our compositional performance analysis approach is the ability to adapt the possible timing of events in an event stream (expressed through the adaptation of an event model [98]) to satisfy requirements imposed on the event stream. Such requirements can have different origins. It may be that a scheduler or a scheduling analysis for a particular component requires certain event stream properties. For example, rate-monotonic scheduling and analysis [68] require strictly periodic task activation. Alternatively, an integrated IP component may require certain event stream properties. External system outputs may also impose event model constraints, e. g. a minimum distance between output events or a maximum acceptable jitter. Such a constraint may be the result of a performance contract with an external subsystem [115].

Event stream adaptation can also be done for the sole purpose of traffic shaping [98]. Traffic shaping can be used e. g. to reduce transient load peaks, in order to obtain more regular system behavior. It is also possible to completely decouple non-functional cyclic scheduling dependencies by reducing event stream jitter to zero. This can assure that global analysis reaches a fix-point that without adaptation would not have been reached (section 3.6.5).

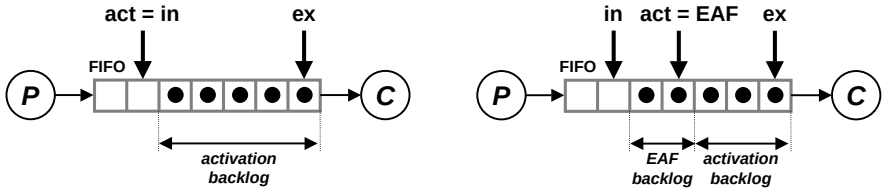
In our compositional performance analysis approach, we logically distinguish *event model interfaces* (EMIFs) and *event adaptation functions* (EAFs) [98]. An EMIF transforms between different event models merely by calculating the target event model parameters from the source event model parameters. Such a transformation is only possible in a few cases. The transformation can be lossless, e. g. from *periodic with jitter* $= 0$ to strictly *periodic* (no jitter parameter) with the same period. It can also be lossy, e. g. from *periodic with jitter* to *sporadic*. If the minimum distance between events in the *periodic with jitter* event model is not smaller than the required minimum distance between events in the target even model, then the target event model is satisfies, but the information about periodicity is lost.

In all other cases, active buffering of events is needed to arrive at the desired result, e. g. from *periodic with jitter* > 0 to strictly *periodic*. An EAF serves this purpose. Practically, we distinguish event model

adaptation from event model *shaping* in SymTA/S [107]. Adaptation is required to satisfy an event model constraint, while shaping is voluntary to obtain more regular system behavior. We have currently implemented two types of EAFs: a *periodic EAF* produces a strictly periodic output event stream from a *periodic with jitter* input event stream. A d_{min}*-EAF* enforces a minimum distance between output events.

3.6.7 Communication Buffers

Two effects determine the required size of communication buffers, as well as the worst- and best-case buffering delay: activation buffering (section 3.6.2) and EAF buffering (section 3.6.6). We assume a FIFO buffer at the input of a task which receives all incoming tokens. Tokens are kept in the buffer from the moment of arrival until the instance of the task that was activated by the token has been completed (section 3.6.1). An optional EAF may delay task activation after the arrival of a token.



(a) Activation buffering

(b) Activation and EAF buffering

Figure 3.8. Communication buffering in our compositional performance analysis approach

Consider the example buffer states in Fig. 3.8 for a buffer between producer task P and consumer task C. Pointer ex (execute) points at the oldest token whose corresponding task instance has not been completed. Pointer in (incoming) points at the FIFO position where the next incoming token will be stored. The distance between in and ex is the current backlog. If the buffer is empty, then both pointers point at the same buffer position, indicating a backlog of zero.

In Fig. 3.8(a), each incoming token immediately leads to a consumer task activation ($act = in$). Consequently, we call the backlog *activation backlog*. In Fig. 3.8(b), an EAF has been additionally inserted at the consumer task input. The EAF determines when a token leads to an activation ($act = EAF$). The EAF pointer follows the in pointer according to the type and parameters of the EAF, e. g. with a minimum distance between steps in case of a d_{min}-EAF. Consequently, the backlog

is separated into *EAF backlog* and *activation backlog*. In these particular buffer states, the distinction between Figs 3.8(a) and 3.8(b) has no effect on task C. Assume however that both the total backlog and the EAF backlog (in Fig. 3.8(b)) drop to 2. In that case, the activation backlog is 2 in Fig. 3.8(a), but 0 in Fig. 3.8(b). Consequently, the task cannot start executing in the presence of the EAF.

3.7 Composition Using Packet Flows and Resource Streams

The compositional performance analysis methodology for network processors developed by Thiele et al. focuses on scheduling analysis of packet flows [109]. Packet flows are described by upper and lower arrival curves [23, 87], which capture minimum and maximum packet-sizes, maximum gaps between packets, peak-load and long-term load. An example is shown in Fig. 3.9.

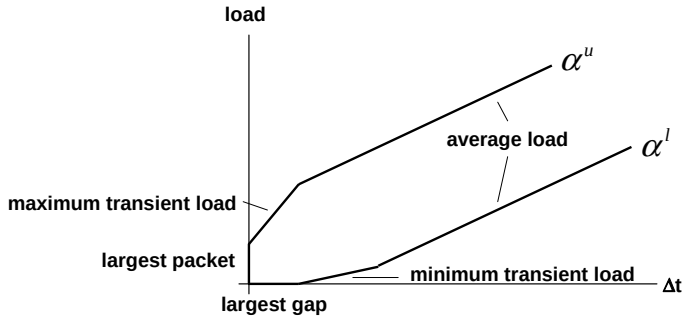


Figure 3.9. Arrival curves used to describe packet flows

Additionally, so called resource streams described by upper and lower service curves represent the available processing power on a particular resource. Each task is modeled with two inputs, one for an *input packet flow*, one for an *available resource stream*, and with two outputs, one for an *output packet flow*, one for a *remaining resource stream*. The arrival curves describing the output packet flow, as well as the service curves describing the remaining resource stream are constructed from the input arrival and service curves using a so called *network calculus*. A nice property of this approach is that the basic network calculus equations for one task are independent of a scheduling policy. Scheduling policies are expressed by appropriately connecting packet flows and resource streams on one resource. An example with two tasks with two different priorities on the same resource is given in Fig. 3.10 (task 1 has a higher priority than task 2).

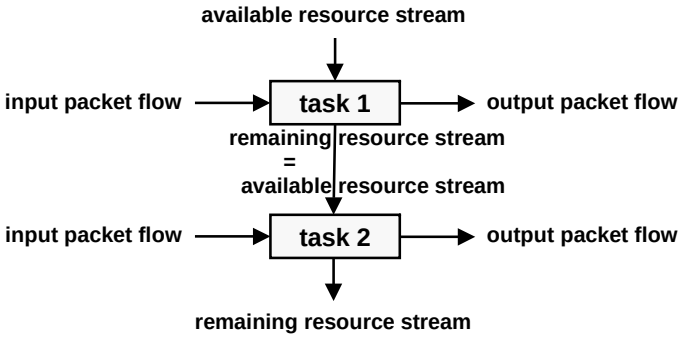


Figure 3.10. Network calculus with two priority-scheduled tasks (task 1 has a higher priority than task 2)

Modeling other schedulers is less straight-forward and may require additional operations beyond the basic network calculus equations [16]. A novel solution has to be found for each scheduler, since the network calculus is different from standard scheduling analysis techniques developed in the real-time community. In particular, this may require considerable work for detailed scheduling analyses taking operating-system overhead, blocking times, or event correlations into account.

Local analyses can be composed into a *scheduling network* [109] with the same flexibility and scalability advantages that our compositional performance analysis has over holistic approaches. Tasks are connected through packet flows in the same way as tasks are connected trough event stream in our approach. Thiele's group has successfully demonstrated how quickly a design-space can be explored using their network calculus. They also evaluated different genetic algorithms to guide the exploration process [109].

A major difference to our approach is the lack of discrete task activations in packet flows. This is reflected in the shape of arrival curves, which are piecewise linear and continuous. Thiele's group has demonstrated that their network calculus is also applicable to discrete event streams [16], and has extended their model to capture different types of events [77]. While such a representation allows a more detailed model of possible event timing compared to our set of event models, it is not clear what kind of additional detail can actually be obtained as a result of scheduling analysis, how complex it is to propagate and evaluate that additional detail, and whether the obtainable benefit justifies the additional complexity and the need to derive scheduling analysis algorithms anew.

3.8 Performance Analysis for Complex Applications

In section 2.1, we gave an overview over a range of models of computation (MOCs), languages and tools used during application specification in embedded system design. It became obvious that embedded applications exhibit a variety of task-dependencies beyond the simple sequential dependencies considered so far by our compositional performance analysis approach 3.5. Specifically, we need to consider multiple activating inputs (with different concatenation semantics), data-rate transitions, conditional communication, cyclic task dependencies and data-dependent behavior.

In this section we summarize to what extent the performance analysis techniques presented in the previous sections support complex task dependencies. A overview is given in Tab. 3.8.

analysis	multiple inputs	rate transitions with fixed rates	conditional communication	cycles	data-dependent behavior
Eles	AND	$\surd$ (unrolling)			
Palencia	AND, OR	$\surd$ (N to 1)			
Thiele (packet flow)	OR				
Thiele (event streams)					$\surd$
Richter					

Table 3.2. Complex application properties supported by a variety of performance analysis approaches

The holistic approach by Eles et al. [90–92] supports AND-activation and data rate transitions. The latter is supported through unrolling into a single-rate schedule. The same approach has been proposed by Ziegenbein [127]. Unrolling is common for static scheduling of SDF graphs [62]. However, unrolling makes scheduling analysis of preemptive systems considerably more difficult, since the length of the unrolled time-window has to be considered in each step. It is also not applicable to data rate transitions, where the producer or consumer data rate is an interval.

The holistic approach by Palencia et al. supports AND- and OR-activation as well as data-rate transitions, but only if external input events are strictly periodic [86]. In particular, the effect that these acti-

vation dependencies have on activation jitter is not considered. Therefore, their approach is limited to relatively simple, regular systems, and in general is not applicable if subsystem-integration is required (section 3.4). Furthermore, in their approach OR-activation requires duplication of subsystems, which in the worst-case leads to system-growth that is exponential with the number of OR-activated nodes. Finally, allowing only N to 1 rate transitions is a severe restriction since it does not allow cyclic dependencies (see example in section 2.1.1).

Both Eles et al. and Palencia et al. additionally model phase information between the activation of different tasks to calculate tighter analysis bounds. This is also possible in our compositional approach, as we will see in chapter 8.

The network calculus for packet flows by Thiele et al. [109] supports OR-concatenation. Total packet flow into a task is the sum of packet flows at all inputs. Individual activations are not considered. When the network calculus is applied to event streams, then data-dependent behavior can be considered [77], but only for sequential task dependencies.

Our own compositional performance analysis approach (Richter) [97, 98] currently does not consider complex task dependencies. However, the composition of local scheduling analysis techniques for the analysis of arbitrarily complex architectures is very attractive and merits an extension of our approach to complex applications. In the following chapters we will therefore develop solutions to support each of the application features identified above as important for the analysis of real-world embedded applications.

3.9 Other Performance Analysis Techniques

Performance analysis techniques that are not based on scheduling analysis have also been proposed. We take a look at two interesting approaches.

3.9.1 Model Checking-Based Scheduling Analysis

In model checking, a system is modeled as an automaton, and an exhaustive search-algorithm traverses the state-space to prove or disprove certain system properties. Several groups have used model checking based on timed automata [2] to derive solutions for certain scheduling problems. The approach seems quite successful to derive schedules when the timing of task activations is predictable, and when task preemption is not possible. An example is the scheduling of manufacturing jobs on a complex assembly-line [30, 1]. A second example is the validation of telecommunication protocols. In the tool TAXYS [22], non-

preemptive single-processor schedules specified in the synchronous language ESTEREL [9] augmented by uncertainty and timing constraints are validated using a timed automaton. It seems that the strength of model checking based on timed automata lies in the verification of logical properties of a scheduler such as reachability of all scheduler states or deadlock-free operation. These properties cannot be checked by the aforementioned scheduling analysis techniques, which assume a correct scheduler.

When it comes to verification of temporal task properties assuming preemptive scheduling (e. g. response times), then on one hand an inefficiency of timed automata, namely the inability to stop and restart clocks, requires cumbersome workarounds. On the other hand, the well-known state space explosion problem inherent in model checking becomes apparent. A straight-forward model using the model checking tool UPPAAL [118] for as little as 10 tasks sharing one processor with a very simple scheduler and periodic with jitter activation does not complete analysis in any reasonable time [71]. A hybrid approach by Fersman is to combine timed automata with standard task models and scheduling analysis approaches from the real-time community [31]. The search-space is reduced because the task model and the known scheduling analysis approach guide the model checker to the interesting points. This produces results in reasonable time for single-processor scheduling. Fersman compares her approach to scheduling analysis which assumes periodic task activation, and argues that in general activation timing can be more complex, and is thus best modeled using a timed automaton. However, she gives no indication if she has timing in mind beyond periodic, sporadic, jitter and bursts, which are state-of-the-art in scheduling analysis. She also does not consider core execution time intervals or best-cases, both of which are required for the analysis of complex systems. Therefore, while it is interesting to observe that model-checking can be applied to solve certain scheduling-analysis problems, the future direction of this work is unclear. In particular, we are not aware of any publications on model checking-based scheduling analysis of complex multi-processor systems.

3.9.2 Tightly Coupled System Model and Scheduling Analysis

Several tool-flows have been proposed where a specific application representation is tightly coupled with a specific set of analysis techniques. An interesting example is the RADHA-RATAN tool-chain by Gupta et al. [24, 25, 72]. The focus of this tool-chain is the calculation of possible execution rates for communicating tasks in a reactive real-time system. These rates can then be compared against timing constraints. For this

purpose, a system is represented as a *generalized task graph* using a simple specification language. Generalized in the sense that tasks with different consumer and producer data-rates are allowed, similar to synchronous dataflow (section 2.1.1), as well as complex topologies including cycles. Generalized task graphs have a two-level hierarchy, where the upper level is cycle-free, and each of its nodes can be a strongly connected component (SCC) on the lower level. The hierarchy is mandated by the fact that the rate derivation techniques for acyclic and cyclic components are different [24, 25].

Rate derivation for acyclic task graphs is part of RADHA [24, 25], while RATAN focuses on cyclic task graphs [24, 72]. RATAN is more restricted than RADHA since it allows only single-rate cycles, and only a subset of the concatenation rules for tasks with multiple inputs.

RADHA-RATAN supports a variety of activation conditions for tasks with multiple inputs. Specifically, these are AND- and OR-activation, which are further refined into un-skipped and skipped tasks. An un-skipped task has to consume every token produced by one of its predecessors. A skipped task is allowed to drop tokens if it cannot keep up with the rate of token arrivals. The expressiveness of the task graphs is the most interesting feature of RADHA-RATAN from our perspective, since it allows to model realistic embedded applications.

On the downside, while the authors claim that RADHA-RATAN is a hardware/software-codesign methodology [24], SW synthesis remains unclear. This is due to the *independence assumption* for tasks [72], which implies that each task has its own exclusive resource. While this can be a valid assumption for hardware-synthesis, it is unrealistic for software, where multiple tasks share a processor under the control of an operating system. Resource sharing is not considered in RADHA-RATAN. Therefore, there is no obvious application of this approach to scheduling analysis of complex embedded applications executed on heterogeneous architectures with RTOSes and bus-arbitration.

3.10 Summary and Conclusion

In this chapter, we gave an overview of the state-of-the-art in formal system-level performance analysis. Scheduling analysis, which calculates worst- and best-case response times for tasks sharing a single component, forms the basis of sophisticated performance analysis approaches for multi-processor architectures. We focused on a compositional performance analysis approach developed by our group which allows to compose the large number of existing scheduling analysis techniques, is applicable to complex, heterogeneous architectures, and supports subsystem integration.

One problem of scheduling analysis is that the calculated corner cases can be overly conservative, because scheduling analysis may not model all scheduling effects in sufficient detail. Analysis tightness can be improved considerably by modeling and evaluating select additional information about the application (so called contexts). Our contribution to analysis improvements through the use of contexts will be presented in chapter 8.

However, the main drawback of our compositional performance analysis approach is that it considers only single-rate, single-input tasks without cyclic dependencies. As explained in the previous chapter, such task graphs are too simple for realistic embedded applications. These shortcoming are remedied in the following three chapters in order to enable compositional performance analysis for complex applications executed on heterogenous architectures.

Chapter 4

TASKS WITH MULTIPLE INPUTS

Our compositional performance analysis approach (section 3.5) assumes that one producer task can be connected to exactly one consumer task, and vice versa. Consequently, the only application topology supported are functionally independent chains of tasks[1].

In realistic embedded applications, task activation often depends on tokens arriving from multiple other tasks and consequently requires multiple inputs. Likewise, a task often sends (potentially different) tokens to multiple tasks and therefore may require multiple outputs. Multiple inputs are the more complicated situation, since the possible activation timing of a multi-input task depends on the possible output timing of all tasks that are sending tokens.

Definition 4.1 (Input Event Functions) *A lower input event function $\eta_i^l(\Delta t)$ specifies the minimum number of times tokens become available for consumption at a consumer task port i during any time interval of length Δt. An upper input event function $\eta_i^u(\Delta t)$ specifies the maximum number of times tokens become available for consumption at a consumer task port i during any time interval of length Δt.*

In the following, the dependency of η_i^l and η_i^u on Δt is omitted for brevity.

The activation function of a task is a boolean function of input events at the different task inputs. A restriction is that activation must not

[1] A chain could also form a loop, but then would have neither an external input for activation, nor an external output to observe results. Nevertheless, it is possible to model such a 'useless' loop. In that case, our compositional performance analysis approach will not work, since an activating event model for the tasks in the loop cannot be determined.

be invalidated due to the arrival of additional tokens [123]. This means that negation is not allowed in the activation function. Consequently, the only acceptable boolean operators are *AND* and *OR*, since an input is negated in all other commonly used boolean operators (NOT, XOR, NAND, NOR)[2]. In practice, this is not a restriction since negation does not occur directly in commonly used MOCs. Negation internally would have to be implemented by a separate 'activator' task that evaluates the activation function upon arrival of a token to decide if activation of the main task should be revoked. The 'activator' task would need to be OR-activated.

AND- and OR-activation will be considered individually first. In section 4.3, some issues that arise from combination of AND- and OR-activation will be discussed.

4.1 AND-Activation

For a consumer task C with multiple inputs, AND-activation implies that C is activated if tokens are available for consumption at each input i. An example of an AND-activated task is shown in Fig. 4.1.

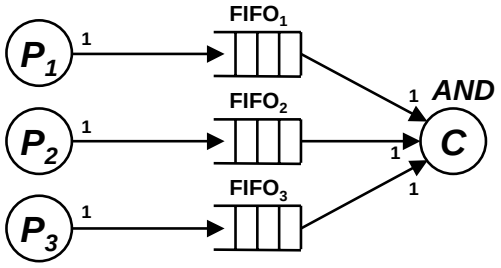


Figure 4.1. Example of an AND-activated task C with three inputs

Input tokens may have to wait at some inputs until enough tokens have arrived at all other inputs for one consumer activation. This requires token buffering, which we will refer to as *AND-buffering*. AND-buffering is required on top of activation buffering (section 3.6.7). Logically, the former holds tokens before task activation. The latter holds tokens from the moment of task activation until execution of the task has been completed. We will consider token buffering due to a combination of both effects in section 5.5. In this section, we concern ourselves exclusively with AND-buffering.

[2]Using the same logic, buffered one-to-many communication is not allowed, since the activation of one task could be invalidated through the removal of an input token from the shared buffer by another task.

To ensure bounded AND-buffer sizes, the following requirements are sufficient.

1 For $\lim_{\Delta t \to \infty}$, the difference between the minimum and maximum number of input events at input i must be finite.

2 For $\lim_{\Delta t \to \infty}$, the difference between the number of input events at *different* inputs must be finite.

Together, both requirements guarantee that every arriving token will lead to a consumer activation in finite time. Note that the first requirement can be relaxed if the numbers of input events at different inputs are suitably correlated. Such correlations will be briefly discussed in chapter 8. Without a model of such correlations the first requirement in particular implies that sporadic input events are not acceptable.

4.1.1 Calculation of Activating Event Functions

Returning to our example, imagine that tasks $P1$, $P2$, $P3$ and C are mapped onto resources and scheduled together with other tasks under RTOS control. In order to perform scheduling analysis on the resource to which C is mapped, activating event functions for task C have to be calculated from the input event functions at the three inputs.

Let us assume for now that task C requires one token at each input for one activation. This restriction will be dropped in section 5.3, after we have discussed data rates and data rate transitions. During any time interval Δt, the port with the smallest minimum number of available tokens determines the minimum number of AND-activations. Likewise, the port with the smallest maximum number of available tokens determines the maximum number of AND-activations. Intuitively, one could think that the maximum number of AND-activations is determined by the port with the smallest upper input event function. However, the number of available tokens at input port i during a time interval Δt depends on both the number of tokens arriving during Δt, and on the number of tokens that arrived earlier, but did not yet lead to an activation because tokens at one or more other input ports are still missing. This is illustrated in the following example. Let us assume that our task in Fig. 4.1 receives tokens at each input with the following *periodic with jitter* input event models:

$$\begin{aligned} \mathcal{P}_1 &= 4, \quad \mathcal{J}_1 = 0 \\ \mathcal{P}_2 &= 4, \quad \mathcal{J}_2 = 2 \\ \mathcal{P}_3 &= 4, \quad \mathcal{J}_3 = 3 \end{aligned}$$

Fig. 4.2 shows a possible sequence of input events that adhere to these event models, and the resulting AND-activation events. The numbering of events in the figure indicates which events together lead to one activation of AND-activated task C.

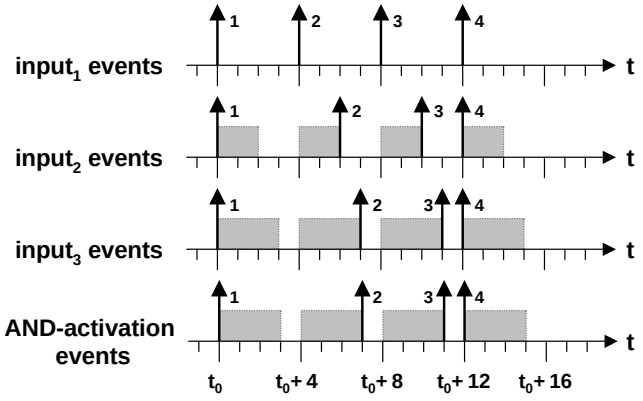


Figure 4.2. AND-activation timing example for a valid sequence of input events

As can be seen, the minimum distance between two AND-activations (activations 3 and 4 in Fig. 4.2) equals the minimum distance between two input events at input 3, which is the input with the largest input event model jitter. Likewise, the maximum distance between two AND-activations (activations 1 and 2 in Fig. 4.2) equals the maximum distance between two input events at input 3. It is not possible to find a different sequence of input events leading to a smaller minimum or a larger maximum distance between two AND-activations. From this we can conclude that the input with the largest input event jitter determines the activation jitter of the AND-activated task. This observation can be generalized to any combination of input event models which satisfy the two AND-activation conditions given above.

LEMMA 4.2 *Let C be an AND-activated task with upper and lower input event functions η_i^u, η_i^l for each input i. Then for any time interval Δt, the upper activating event function of task C equals the maximum of all upper input event functions. Likewise, the lower activating event function of task C equals the minimum of all lower input event functions.*

$$\eta_{AND}^u = \max\{\eta_i^u\} \tag{4.1}$$
$$\eta_{AND}^l = \min\{\eta_i^l\} \tag{4.2}$$

PROOF 4.2 Upon arrival, a token either has to wait for other tokens to arrive for one consumer activation, or it is the last token required for one

consumer activation. Let us assume that a token arrives at input i at time $t_{act,i}$, and that this token is the last token required for one consumer activation. Then η_i^u is an upper bound on the number of times that the input condition at input i can be satisfied during any time interval $\Delta t_{act,i}$ *starting at* $t_{act,i}$, and thus bounds the maximum number of times that task C can be activated during $\Delta t_{act,i}$.

Let us now consider all points in time t_{act} when a token arrival leads to a consumer activation. Without further information we have to assume that the last required token can arrive at any input i. Then, $\max\{\eta_i^u\}$ is an upper bound on the number of times that task C can be activated during a time interval Δt_{act} starting at *any* point in time t_{act}. For all other points in time, the upper bound on the number of times that task C can be activated during a time interval Δt will be equal or lower, since the first activation does not happen until the next t_{act}. This proves equation 4.1.

For any time interval of length Δt, η_i^l is a lower bound on the number of times that the input condition at input i has to be satisfied. Without further information we have to assume that the last required token can arrive at any input i. Then, $\min\{\eta_i^l\}$ is a lower bound for the number of times that task C has to be activated. This proves equation 4.2. □

Note that in Fig. 4.2 we assumed that corresponding input events can occur at the same time. In some cases it may be possible to calculate phases between the arrival of corresponding tokens in more detail, e. g. through the use of inter-contexts (section 8.2). It may then be possible to calculate tighter activating event functions if it can be shown that a certain input cannot (fully) influence the activation timing of an AND-activated task, because tokens at this input arrive relatively early. This is particularly important for the analysis of functional cycles (chapter 6).

4.1.2 AND-Activation Incurred Delay and Backlog

In embedded real-time systems, meeting deadlines as well as accurately dimensioning memory are critical. Therefore, we need to calculate the maximum AND-delay, as well as the maximum AND-backlog and consequently the required AND-buffer size for each input. For this purpose, the lower activating event function is interpreted as a lower service function [109] of a hypothetical AND-concatenation task with 0 execution time: this task serves each input immediately after enough tokens have arrived at all inputs.

This interpretation allows us to reuse results by Thiele et al. who showed in [109] that the maximum delay experienced by input data before it is processed by a consumer task is the maximum *horizontal* distance between the upper arrival curve of the data and the lower service curve of the task. The maximum buffer size required to buffer that data is determined by the maximum *vertical* distance between the upper arrival curve and the lower service curve.

Inserting η_i^u and η_{AND}^l into the delay and backlog inequations from [109] yields

$$\text{delay}_i \leq \max_{\Delta t \geq 0} \left\{ \min_{t \geq 0} \left\{ t \,|\, \eta_i^u(\Delta t) = \eta_{AND}^l(\Delta t + t) \right\} \right\} \tag{4.3}$$

$$\text{backlog}_i \leq \max_{\Delta t \geq 0} \left\{ \eta_i^u(\Delta t) - \eta_{AND}^l(\Delta t) \right\} \tag{4.4}$$

The minimum delay and backlog at each input are zero, since the last token required for AND-activation can arrive at any input.

4.1.3 AND-activation for *periodic with jitter* Input Event Models

For the reasons stated in section 3.1.1 we are specifically interested in *periodic with jitter* input event models, which we describe with parameters ($\mathcal{P}_i$, $\mathcal{J}_i$). We would like to calculate the activating event model ($\mathcal{P}_{AND}$, $\mathcal{J}_{AND}$) for the AND-activated task, which we will need for scheduling analysis.

4.1.3.1 AND-Activation Period

To ensure bounded input buffer sizes, the period of all input event models must be the same. The period of the activating event model equals this period.

$$\begin{aligned} \mathcal{P}_i &\stackrel{!}{=} \mathcal{P}_j \quad ; \quad i,j = 1..k \quad \Rightarrow \\ \mathcal{P}_{AND} &= \mathcal{P}_i \quad ; \quad i = 1..k \end{aligned} \tag{4.5}$$

4.1.3.2 AND-Activation Jitter

The input event model with the largest jitter has both the maximum upper and the minimum lower input event function, and thus according to equations 4.1 and 4.2 determines the jitter of the activating event model of the AND-activated task.

$$\mathcal{J}_{AND} = \max\{\mathcal{J}_i\} \quad ; \quad i = 1..k \tag{4.6}$$

4.1.3.3 AND-Activation Incurred Token Delay and Backlog

The input event model with the largest jitter determines η^l_{AND}, which is then inserted into inequations 4.3 and 4.4.

$$\begin{aligned} \text{delay}_i &\leq \max_{\Delta t \geq 0} \Bigg\{ \\ &\quad \min_{t \geq 0} \left\{ t \,|\, \left\lceil \frac{\Delta t + \mathcal{J}_i}{\mathcal{P}} \right\rceil = \max\left(0, \left\lfloor \frac{\Delta t + t - \mathcal{J}_{AND}}{\mathcal{P}} \right\rfloor\right) \right\} \Bigg\} \end{aligned} \tag{4.7}$$

$$\text{backlog}_i \leq \max_{\Delta t \geq 0} \left\{ \left\lceil \frac{\Delta t + \mathcal{J}_i}{\mathcal{P}} \right\rceil - \max\left(0, \left\lfloor \frac{\Delta t - \mathcal{J}_{AND}}{\mathcal{P}} \right\rfloor\right) \right\} \tag{4.8}$$

To find a simple solution for inequation 4.7, let us first consider the strictly periodic case where all input jitters are zero. In that case, inequation 4.7 becomes

$$\text{delay}_i \leq \max_{\Delta t \geq 0} \left\{ \min_{t \geq 0} \left\{ t \,|\, \left\lceil \frac{\Delta t}{\mathcal{P}} \right\rceil = \left\lfloor \frac{\Delta t + t}{\mathcal{P}} \right\rfloor \right\} \right\}$$

For $\Delta t = n * \mathcal{P} + \lim_{\epsilon \to +0}$; $n \in \mathbb{N}^0$ the required value for t approaches $\mathcal{P}$. For all other Δt, the required value for t is smaller. We conclude that for this simplest case, $\text{delay}_i \leq \mathcal{P}$. This result is also obvious from Fig. 4.3(a) which shows the upper and lower event functions of a strictly periodic event model with period $\mathcal{P} = 4$. Obviously, the maximum *horizontal* distance between the two curves approaches $\mathcal{P}$.

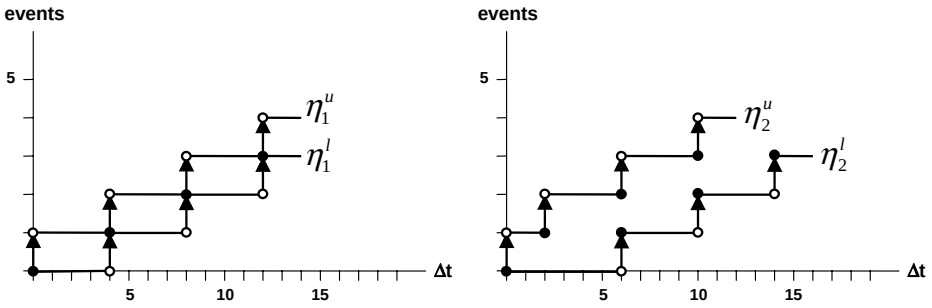
Let us now return to the general case. Compared to the strictly periodic case, a jitter $\mathcal{J}_i$ at input i moves the upper input event function η^u_i left by $\mathcal{J}_i$. The activation jitter $\mathcal{J}_{AND}$ moves the lower activating event function η^l_{AND} right by $\mathcal{J}_{AND}$. Consequently, the maximum horizontal distance between the two functions is

$$\text{delay}_i \leq \mathcal{P} + \mathcal{J}_i + \mathcal{J}_{AND} \tag{4.9}$$

The maximum backlog is obviously incurred for a Δt just before the lower activating event function steps. The lower activating event function steps for the first time at $\Delta t = \mathcal{P} + \mathcal{J}_{AND}$. At that point, the value of the upper input event function is $\lceil (\mathcal{P} + \mathcal{J}_{AND} + \mathcal{J}_i)/\mathcal{P} \rceil$ according to equation 3.1. Since thereafter both functions step periodically every $\mathcal{P}$, we do not need to consider any other point. Consequently

$$\text{backlog}_i \leq \left\lceil \frac{\mathcal{P} + \mathcal{J}_i + \mathcal{J}_{AND}}{\mathcal{P}} \right\rceil \tag{4.10}$$

These inequations can be further improved for the input with the largest jitter, since a token arriving at that input can only wait for



(a) AND input 1 (b) AND input 2

Figure 4.3. Upper and lower input event functions at the first and second input in our AND-example

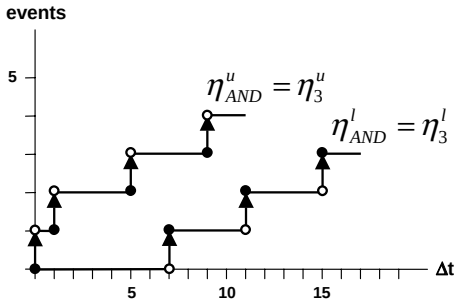


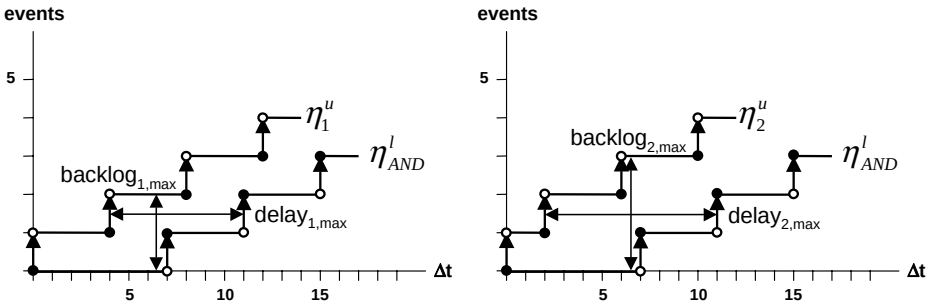
Figure 4.4. Upper and lower input event functions at the third input, as well as upper and lower activating event functions in our AND-example (AND activation $\triangleq$ AND input 3)

tokens arriving at other inputs. Consequently, for the input with the largest jitter, the second largest jitter can be used instead of $\max\{\mathcal{J}_i\}$ in inequation 4.6. If two or more inputs have the same largest jitter, then one of them can be interpreted as the second largest, and a special treatment of these inputs is not necessary.

4.1.4 Example

Let us return to our example in Fig. 4.1. The input event functions as well as the activating event functions are shown in Figs. 4.3 and 4.4. The activating event model equals the input event model at the third input, since it has the largest jitter.

$$\mathcal{P}_{AND} = \mathcal{P}_3 = 4, \quad \mathcal{J}_{AND} = \mathcal{J}_3 = 3$$



(a) AND buffer at input 1

(b) AND buffer at input 2

Figure 4.5. Maximum delay and backlog incurred at the first and second input of our example AND-activated task

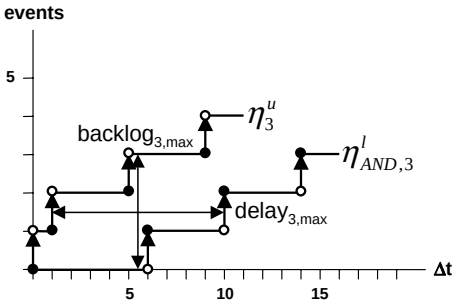


Figure 4.6. Maximum delay and backlog incurred at the third input of our example AND-activated task

We obtain the following values for maximum delay and maximum backlog at each input.

$$\begin{aligned} \text{delay}_1 &\leq 7 \text{ time units}, \quad \text{backlog}_1 \leq 2 \text{ events / tokens} \\ \text{delay}_2 &\leq 9 \text{ time units}, \quad \text{backlog}_2 \leq 3 \text{ events / tokens} \\ \text{delay}_3 &\leq 9 \text{ time units}, \quad \text{backlog}_3 \leq 3 \text{ events / tokens} \end{aligned}$$

The calculation of theses values is shown in Figs. 4.5 and 4.6. Note that we use a different lower service function $\eta_{AND,3}^l$ at the input with the largest jitter (the 3rd input) according to section 4.1.3.3. Further note that backlog and delay at the two inputs with the largest and second largest jitter (inputs 2 and 3) are the same. This is always the case since the left-shift of η_i^u from the second-largest to the largest jitter is

compensated by the same-size left-shift of η^l_{AND} from the largest to the second-largest jitter.

4.2 OR-Activation

For a consumer task C with multiple inputs, OR-activation implies that C is activated each time tokens are available for consumption at any input. Different to AND-activation, input event models are not restricted, and no OR-buffering is required, since tokens at one input never have to wait for tokens to arrive at a different input in order to activate C. Of course, activation buffering (section 3.6.7) is still required. We will consider activation buffering in the presence of OR-activation in section 5.5.

An example of an OR-activated task is shown in figure 4.7.

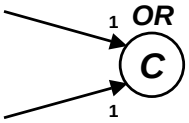


Figure 4.7. Example of an OR-activated task with two inputs

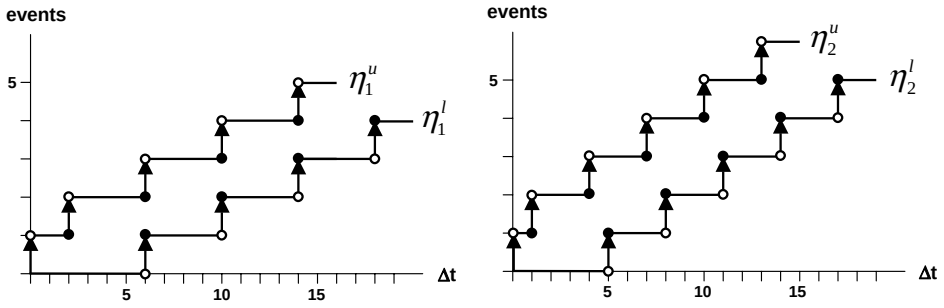
4.2.1 Calculation of Activating Event Functions

We assume that no input event is ever skipped. Therefore, for any Δt, the maximum (minimum) number of activating events for OR-activated task C is the sum of the maximum (minimum) number of input events.

$$\eta^u_{OR} = \sum_{i=1}^{n} \eta^u_i \tag{4.11}$$

$$\eta^l_{OR} = \sum_{i=1}^{n} \eta^l_i \tag{4.12}$$

Formalisms have also been proposed, e. g. by Gupta et al. ([24] and section 3.9.2), that allow to skip input events of OR-activated tasks. Some operating systems, e. g. ERCOSEK [29], skip input events of an OR-activated task as long as the task has not completed its previous execution. This implies that the activating event functions can no longer be calculated from input event functions alone, but that task response times have to be considered. Essentially it means that the upper activating event function is limited from above by a step function, were the distance between steps corresponds to the minimum response time of the activated task. The lower activating event function is limited from



(a) OR input 1 ($\mathcal{P}_1 = 4, \quad \mathcal{J}_1 = 2$)

(b) OR input 2 ($\mathcal{P}_2 = 3, \quad \mathcal{J}_2 = 2$)

Figure 4.8. Upper and lower input event functions in our OR-example

below by a step function, were the distance between steps corresponds to the maximum response time of the activated task. Skipped input events are not further considered in this thesis.

4.2.2 OR-activation for *periodic with jitter* Input Event Models

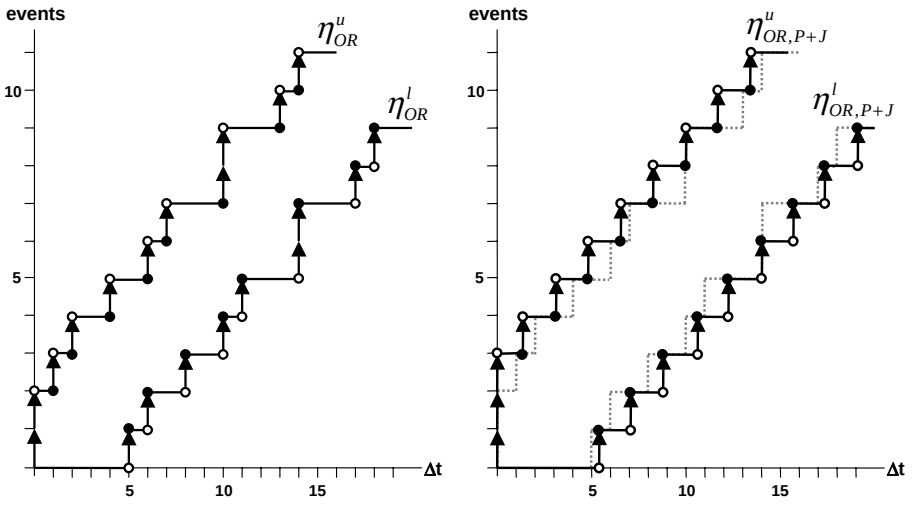
For the reasons stated in section 3.1.1 we are specifically interested in *periodic with jitter* input event models, which we describe with parameters ($\mathcal{P}_i$, $\mathcal{J}_i$). We would like to calculate the activating event model ($\mathcal{P}_{OR}$, $\mathcal{J}_{OR}$) for the OR-activated task, which we will need for scheduling analysis.

Let us consider the example task in Fig. 4.7. Let us assume that the tokens arrive at the two inputs with the following *periodic with jitter* event models:

$$\mathcal{P}_1 = 4, \quad \mathcal{J}_1 = 2$$
$$\mathcal{P}_2 = 3, \quad \mathcal{J}_2 = 2$$

The corresponding upper and lower input event functions are shown in Fig. 4.8. According to equations 4.11 and 4.12, the upper and lower activating event functions for task C are constructed by adding the respective input event functions. The result is shown in Fig. 4.9(a).

Recall a key requirement of compositional performance analysis, namely that event streams are described in a form that can serve both as input for local scheduling analysis, and can be produced as an output of local scheduling analysis for propagation to the next analysis component (section 3.1.1). Due to the irregularly spaced steps (visible in Fig. 4.9(a)),



(a) exact (b) *periodic with jitter* approximation

Figure 4.9. Upper and lower activating event functions in our OR-example

the *exact* activating event functions cannot be described by a *periodic with jitter* event model, and thus cannot serve directly as input for local scheduling analysis. One solution would be to use a scheduling analysis algorithm that considers each *periodic with jitter* input event model individually and internally calculates the sums in equations 4.11 and 4.12. Such an approach would yield the most accurate response times. However, we cannot expect that every analysis algorithm is able to do that.

Furthermore, after local scheduling analysis a *periodic with jitter* output event model has to be propagated to the next analysis component. We need an activation jitter in order to calculate an output jitter (section 3.6.2). Therefore, we need to find conservative approximations for η^u_{OR} and η^l_{OR} that can be described by a *periodic with jitter* event model $(\mathcal{P}_{OR}, \mathcal{J}_{OR})$. The intended result is shown in Fig. 4.9(b) (the exact curves appear as dotted lines).

4.2.2.1 OR-Activation Period

The period of OR-activation is the least common multiple LCM(P_i) of all input event model periods (the *macro period*), divided by the sum of input events during the macro period assuming zero jitter for all input

event streams.

$$\mathcal{P}_{OR} = \frac{\mathrm{LCM}(P_i)}{\sum_{i=1}^{n} \frac{\mathrm{LCM}(P_i)}{\mathcal{P}_i}} = \frac{1}{\sum_{i=1}^{n} \frac{1}{\mathcal{P}_i}} \tag{4.13}$$

4.2.2.2 OR-Activation Jitter

A conservative approximation of η^u_{OR} and η^l_{OR} with a *periodic with jitter* event model implies that equations 4.11 and 4.12 are replaced by the following inequations.

$$\left\lceil \frac{\Delta t + \mathcal{J}_{OR}}{\mathcal{P}_{OR}} \right\rceil \geq \sum_{i=1}^{n} \left\lceil \frac{\Delta t + \mathcal{J}_i}{\mathcal{P}_i} \right\rceil \tag{4.14}$$

$$\max\left(0, \left\lfloor \frac{\Delta t - \mathcal{J}_{OR}}{\mathcal{P}_{OR}} \right\rfloor\right) \leq \sum_{i=1}^{n} \max\left(0, \left\lfloor \frac{\Delta t - \mathcal{J}_i}{\mathcal{P}_i} \right\rfloor\right) \tag{4.15}$$

In order to be as accurate as possible, we are interested in the minimum jitter that satisfies inequations 4.14 and 4.15. Let us define $\mathcal{J}^u_{OR}$ to be the minimum upper activation jitter that satisfies inequation 4.14, and $\mathcal{J}^l_{OR}$ to be the minimum lower activation jitter that satisfies inequation 4.15. Then the first interesting question is whether $\mathcal{J}^u_{OR}$ and $\mathcal{J}^l_{OR}$ are the same.

Lemma 4.3 *For any combination of* periodic with jitter *input event models of an OR-activated task C, the minimum upper activation jitter $\mathcal{J}^u_{OR}$ and the minimum lower activation jitter $\mathcal{J}^l_{OR}$ are the same.*

Proof 4.3 Let us first assume that $\mathcal{J}_i = 0$ for all i input event models of task C. Then, the upper and lower activating event functions of task C meet periodically each macro period, i. e. each $\Delta t_{\mathrm{LCM},n}$ where $\Delta t_{\mathrm{LCM},n} = n * \mathrm{LCM}\{\mathcal{P}_i\}\,;\, n \in \mathbb{N}^0$. The activating event functions obviously display an inverse symmetry around each $\Delta t_{\mathrm{LCM},n}$. If the upper activating event function steps at $\Delta t^u = \Delta t_{\mathrm{LCM},n} + t$, then the lower activating event function steps at $\Delta t^l = \Delta t_{\mathrm{LCM},n} - t$.

Upper and lower event function $\eta^u_{OR,\mathcal{P}+0}$ and $\eta^l_{OR,\mathcal{P}+0}$ which correspond to a *periodic with jitter* event model with $(\mathcal{P} = \mathcal{P}_{OR}\,,\, \mathcal{J} = 0)$ also meet periodically each $\Delta t_{\mathrm{LCM},n}$. These two event functions are *not* conservative approximations of η^u_{OR} and η^l_{OR}. To obtain conservative approximations, $\eta^u_{OR,\mathcal{P}+0}$ has to be shifted left by a minimum amount, and $\eta^l_{OR,\mathcal{P}+0}$ has to be shifted right by a minimum amount. These minimum shifts equal $\mathcal{J}^u_{OR}$ and $\mathcal{J}^l_{OR}$, respectively. Due to the aforementioned inverse symmetry, $\mathcal{J}^u_{OR} = \mathcal{J}^l_{OR}$.

Let us now drop the initial assumption that $\mathcal{J}_i = 0$ for all i. The upper and lower activating event functions of task C will no longer meet, but the inverse symmetry around each $\Delta t_{\mathrm{LCM},n}$ remains (e. g. around $\Delta t = 12$ in Fig. 4.9(a)). The reason is that jitter increases the maximum, and decreases the minimum distance between consecutive events by the same amount (equations 3.3 and 3.4). Therefore, $\eta^u_{OR,\mathcal{P}+0}$ and $\eta^l_{OR,\mathcal{P}+0}$ still have to be shifted left respectively right by an equal minimum amount to obtain conservative approximations of η^u_{OR} and η^l_{OR} [3]. Consequently, $\mathcal{J}^u_{OR} = \mathcal{J}^l_{OR}$ is true for any combination of *periodic with jitter* input event models of an OR-activated task. □

In the following, the upper approximation (inequation 4.14) is used to calculate the OR-activation jitter. Since the left and right sides of this inequation are only piecewise continuous, the inequation cannot be simply transformed to obtain the desired minimum jitter. The solution used here is to evaluate inequation 4.14 piecewise for each interval $]\Delta t_j, \Delta t_{j+1}]$, during which the right side of the inequation has a constant value $k_j \in \mathbb{N}$. For each constant piece of the right side, a condition for a *local jitter* $\mathcal{J}_{OR,j}$ is obtained that satisfies the inequation for all $\Delta t : \Delta t_j < \Delta t \leq \Delta t_{j+1}$.

For each constant piece of the right side, inequation 4.14 becomes

$$\left\lceil \frac{\Delta t + \mathcal{J}_{OR,j}}{\mathcal{P}_{OR}} \right\rceil \quad \geq \quad k_j \quad ; \quad \Delta t_j < \Delta t \leq \Delta t_{j+1}\,,\, k_j \in \mathbb{N}$$

Since the left side of this inequation is monotonically increasing with Δt, it is sufficient to evaluate it for the smallest value of Δt, which approaches Δt_j. I. e.

$$\begin{aligned}
& \lim_{\epsilon \to +0} \left\lceil \frac{\Delta t_j + \epsilon + \mathcal{J}_{OR,j}}{\mathcal{P}_{OR}} \right\rceil \;\geq\; k_j \quad ; \quad k_j \in \mathbb{N} \\
\Leftrightarrow \quad & \lim_{\epsilon \to +0} \frac{\Delta t_j + \epsilon + \mathcal{J}_{OR,j}}{\mathcal{P}_{OR}} \;>\; k_j - 1 \\
\Leftrightarrow \quad & \lim_{\epsilon \to +0} (\mathcal{J}_{OR,j} + \epsilon) \;>\; (k_j - 1) * \mathcal{P}_{OR} - \Delta t_j \\
\Leftrightarrow \quad & \mathcal{J}_{OR,j} \;\geq\; (k_j - 1) * \mathcal{P}_{OR} - \Delta t_j \qquad (4.16)
\end{aligned}$$

The global minimum jitter is then the smallest value which satisfies all local jitter conditions. As already said, η^u_{OR} displays a pattern of distances between steps which repeats periodically every macro period. Therefore, it is sufficient to perform above calculation for one macro period. An algorithm can be found in [41].

[3] The curves in Fig. 4.9(b) are the result of this operation.

4.2.3 Example

Let us return to our initial example and calculate the activating event model for OR-activated task C. According to equation 4.13 we obtain the following OR-activation period.

$$\mathcal{P}_{OR} \;=\; \frac{\mathcal{P}_1 * \mathcal{P}_2}{\mathcal{P}_1 + \mathcal{P}_2} \;=\; \frac{12}{7} \;\approx\; 1.714$$

OR-activation jitter is calculated according to inequation 4.14, which becomes

$$\left\lceil \frac{\Delta t + \mathcal{J}_{OR}}{12/7} \right\rceil \;\geq\; \left\lceil \frac{\Delta t + 2}{4} \right\rceil + \left\lceil \frac{\Delta t + 2}{3} \right\rceil$$

Constant segments of the right side of this inequation now have to be found. The first constant segment is for $0 < \Delta t \leq 1$. According to inequation 4.16

$$\left\lceil \frac{\Delta t + \mathcal{J}_{OR,0}}{12/7} \right\rceil \;\geq\; 2 \qquad ;\quad 0 < \Delta t \leq 1$$

$$\Leftrightarrow \qquad \mathcal{J}_{OR,0} \;\geq\; 1 * \frac{12}{7} - 0 \;=\; 12/7$$

The following table shows all constant segments and the resulting local $\mathcal{J}_{OR,i}$ for the first macro period. Each constant segment can also be seen in the plot of η^u_{OR} in Fig. 4.9(a).

Δt range	k_j	local $\mathcal{J}_{OR,j}$
$0 < \Delta t \leq 1$	2	$\mathcal{J}_{OR,0} \geq 1 * \frac{12}{7} - 0 = \frac{12}{7}$
$1 < \Delta t \leq 2$	3	$\mathcal{J}_{OR,1} \geq 2 * \frac{12}{7} - 1 = \frac{17}{7}$
$2 < \Delta t \leq 4$	4	$\mathcal{J}_{OR,2} \geq 3 * \frac{12}{7} - 2 = \frac{22}{7}$
$4 < \Delta t \leq 6$	5	$\mathcal{J}_{OR,3} \geq 4 * \frac{12}{7} - 4 = \frac{20}{7}$
$6 < \Delta t \leq 7$	6	$\mathcal{J}_{OR,4} \geq 5 * \frac{12}{7} - 6 = \frac{18}{7}$
$7 < \Delta t \leq 10$	7	$\mathcal{J}_{OR,5} \geq 6 * \frac{12}{7} - 7 = \frac{23}{7}$
$10 < \Delta t \leq 13$	9	$\mathcal{J}_{OR,6} \geq 8 * \frac{12}{7} - 10 = \frac{26}{7}$
$13 < \Delta t \leq 14$	10	$\mathcal{J}_{OR,7} \geq 9 * \frac{12}{7} - 13 = \frac{17}{7}$

The last table entry only shows that starting with $\Delta t > 1$, the pattern repeats every $\Delta t = \mathrm{LCM}(\mathcal{P}_1, \mathcal{P}_2) = 4 * 3 = 12$ time units.

Finally, we take to largest $\mathcal{J}_{OR,i}$ to obtain the activating event model jitter.

$$\mathcal{J}_{OR} \;=\; \frac{26}{7} \approx 3.714$$

4.2.4 OR-activation for *sporadic with jitter* Input Event Models

If all input event models for an OR-activated task are *sporadic with jitter*, then the same rules to calculate the activating event model apply as for *periodic with jitter* event models. The only difference of course is that the activating event model is also *sporadic with jitter*.

If some input event models are *periodic with jitter* and some are *sporadic with jitter*, then the simplest, but very conservative solution is to treat all event models as *sporadic with jitter*. This is valid, since a *sporadic with jitter* event model is a conservative approximation of a *periodic with jitter* event model with the same parameters [98].

A second solution yielding tighter bounds is to perform a second calculations to obtain separate *period* and *jitter* parameters that more accurately models the lower activating event function of the OR-concatenated task. This calculation works exactly as described above, but ignores all *sporadic with jitter* input event models and considers only the *periodic with jitter* ones. This solution is only useful if our system-level analysis framework is extended to propagate two separate sets of parameters for each event stream. The additional complexity is bounded, since it is never necessary to maintain more than two sets per event stream (one for the upper, and one for the lower event function). During scheduling analysis, worst case load, worst case response time and upper output event function parameters are calculated using the parameters for the upper input event function, and best case load, best case response time and lower output event function parameters are calculated using the parameters for the lower input event function.

4.3 Combination of AND- and OR-Activation

A task may have an activation function with nested AND- and OR-concatenations, e. g. $(i1 \textit{ OR } i3) \textit{ AND } i2$. For each concatenation, the concatenated event model is calculated as described in sections 4.1 and 4.2, and used as an input event model for the next concatenation. The number of nested concatenations, as well as the order of AND- and OR-concatenations is not restricted. As was the case for pure AND-activation, it has to be ensured that buffer over- or underrun can be reliably avoided in the presence of a combination of OR- and AND-activation. This implies that all input event models for each AND-concatenation must be periodic and have the same period. For our example, we obtain the following condition:

$$\frac{\mathcal{P}_1 * \mathcal{P}_3}{\mathcal{P}_1 + \mathcal{P}_3} \stackrel{!}{=} \mathcal{P}_2$$

An interesting problem arises if the same input appears more than once in the activation function. For example, the following is a correct boolean transformation:

$$(i1 \textit{ OR } i3) \textit{ AND } i2 = (i1 \textit{ AND } i2) \textit{ OR } (i2 \textit{ AND } i3)$$

This implies $\mathcal{P}_1 \stackrel{!}{=} \mathcal{P}_2 \stackrel{!}{=} \mathcal{P}_3$, which is obviously not consistent with the condition that we gave above. The reason is that we implicitly assumed that $i2$ can be read twice. Since this is not possible in a FIFO, we conclude that an input must not appear more than once in the activation function.

4.4 Multiple Outputs

A different number of tokens can be produced at different outputs of a multi-output task at the end of one execution. The output rate can be a fixed number of tokens or an interval (including a lower bound of zero). Data rates will be considered in detail in the next chapter. However, we assume that the specified number (interval) of tokens is produced at each output every execution. Tasks that communicate conditionally are modeled with an output rate interval. Once we consider intra-contexts during analysis, it starts to make sense to explicitly distinguish different task behaviors modes. We will return to this in section 8.1.

4.5 Summary and Conclusion

In this chapter we showed how to apply our compositional performance analysis approach to tasks with multiple activating inputs. We argued that AND- and OR-activation (and their combination) are the only meaningful concatenations of multiple activating inputs. We calculated activating event models for both types of concatenations, as well as required communication buffers and incurred token delay for AND-activation. In case of OR-concatenation, we had to conservatively approximate the exact activating event functions, in order to obtain period and jitter parameters required in our compositional performance analysis approach.

We used input event functions which allowed us to abstract from the specific number of tokens required at a particular input for one consumer task activation. For simplicity, we assumed a single required token per input during buffer-size calculation. In the following chapter we will first consider data rate transitions between tasks, and then combine those results with the results from this chapter into a general model which allows to calculate activating event models and the required buffers in the presence of both data rate transitions and multiple activating inputs.

Chapter 5

RATE TRANSITIONS BETWEEN TASKS

So far, we have assumed that a task consumes (produces) one token per execution per input (output). However, in realistic applications a task may consume (produce) *multiple tokens* per execution at a particular input (output). From this follows that the number of tokens produced by a producer task at a particular output may be different from the number of tokens consumed by a consumer task at the connected input, leading to a *data rate transition*. Furthermore, task communication may also be conditional, leading to data rate transitions with *intervals*.

Data rate transitions occur in MoCs where process activation depends on the availability of a certain amount of data (section 2.1.1). A prominent example of such dataflow models is Synchronous Dataflow (SDF) [61], which is a standard MoC used in signal-processing. For example, Fig. 5.1 shows an SDF model of a chain of filters which convert the digital audio format used for CDs to one used for DATs (digital audio tapes).

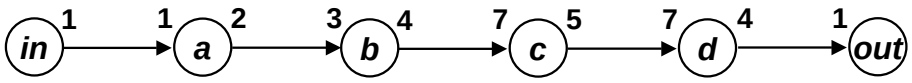


Figure 5.1. SDF model of a chain of filters which convert digital audio from CD to DAT format. The system exhibits data rate transitions between tasks.

Data rate transitions have two implications. They change the frequency of consumer activations compared to producer executions. They may also require buffering of some tokens (rate transition buffering) until enough tokens have arrived for one consumer activation [61]. Note that

a rate transition buffer is required on top of the execution buffer that hold tokens from the moment of task activation until execution of the task has been completed (section 3.6.7). We will consider token buffering due to a combination of these effects in section 5.5. In this chapter, we concern ourselves exclusively with rate transition buffering.

We first consider data rate transitions between fixed data rates. We initially assume tasks with only one input and one output. We show how to calculate activating event functions from output event functions in the presence of a data rate transition. We also calculate the required buffer size and worst-case token delay due to a data rate transition. The results are then extended to data rate transitions with intervals. At the end of the chapter, we consider tasks with multiple inputs and combine data rate transitions with AND- and OR-activation.

5.1 Fixed Data Rates

In a data rate transition with fixed data rates, a producer task produces a fixed number of tokens per execution, the producer data rate r_p. The connected consumer task consumes a different fixed number of tokens per execution, the consumer data rate r_c.

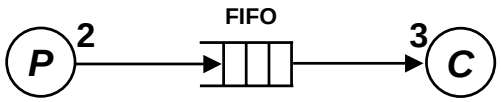


Figure 5.2. Data rate transition with smaller producer data rate ($r_p = 2$) and larger consumer data rate ($r_c = 3$)

Consider the data rate transition example in Fig. 5.2. Task P produces 2 tokens per execution, task C consumes 3 tokens per execution. Now imagine that P and C are mapped onto resources and scheduled together with other tasks under RTOS control. In order to perform scheduling analysis on the resource to which C is mapped, activating event functions for task C have to be calculated from the output event functions of task P. The answer is not obvious, since due to the data rate transition different tokens have to wait for different lengths of time until enough tokens have arrived for one activation of C.

5.1.1 Token Functions

So far, we have implicitly assumed that an output event corresponds to the production of one token, and that an input event corresponds to the arrival of one token. In particular, we have made these assumptions

in the previous chapter when calculating activating event functions for AND- and OR-activated tasks.

In this chapter, the amounts of data produced and consumed per execution are different or even intervals. We introduce producer and consumer *token functions* for the purpose of explicitly modeling the amounts of data produced and consumed data. We initially assume tasks with only one input and one output.

Definition 5.1 (Producer Token Functions) *The lower producer token function $\tau_p^l(\Delta t)$ specifies the minimum number of tokens that task P produces during any time interval of length Δt. The upper producer token function $\tau_p^u(\Delta t)$ specifies the maximum number of tokens that task P can produce during any time interval of length Δt.*

Definition 5.2 (Consumer Token Functions) *The lower consumer token function $\tau_c^l(\Delta t)$ specifies the minimum number of tokens that become available for consumption by task C during any time interval of length Δt. The upper consumer token function $\tau_c^u(\Delta t)$ specifies the maximum number of tokens that become available for consumption by task C during any time interval of length Δt.*

In the following, the dependency of τ^l and τ^u on Δt is omitted for brevity.

The production of r_p tokens is idealized to take zero time. Therefore, producer token functions are piecewise constant between vertical steps whose *height corresponds to the producer's data rate*. The consumer is activated if at least r_c tokens are available. Therefore, consumer token functions are piecewise constant between vertical steps whose *height corresponds to the consumer's data rate*. Token functions are similar to arrival curves [23, 110, 87] with the special property that production and consumption of tokens is atomic and timeless [49, 51].

A simple relationship exists between token functions and event functions. Let η_p^u and η_p^l explicitly refer to the upper and lower event functions describing the output timing of producer task P. Let η_c^u and η_c^l explicitly refer to the upper and lower event functions describing the activation timing of consumer task C. Then

$$\tau_p^u = r_p * \eta_p^u \tag{5.1}$$

$$\tau_p^l = r_p * \eta_p^l \tag{5.2}$$

$$\tau_c^u = r_c * \eta_c^u \tag{5.3}$$

$$\tau_c^l = r_c * \eta_c^l \tag{5.4}$$

If the producer data rate r_p is different from the consumer data rate r_c, then producer and consumer token function have an added level of

expressiveness compared to event functions. This is exploited to calculate activating event functions for the consumer task in the presence of data rate transitions.

5.1.2 Calculation of Activating Event Functions

As already said, the goal of this section is to calculate activating event functions η_c^u and η_c^l from output event functions η_p^u and η_p^l in the presence of a data rate transition between producer task P and consumer task C.

Let us assume that P's output event functions are given. In the first step, *producer* token functions are obtained from P's output event functions using equations 5.1 and 5.2. In the second, central step, *consumer* token functions have to be constructed from producer token functions. Remember that consumer token functions describe the availability of multiples of r_c tokens as a function of Δt.

To correctly construct consumer token functions, all possible numbers of tokens buffered between tasks P and C have to be considered. Therefore, to correctly construct the upper consumer token function, the maximum initial number of tokens in the buffer between P and C not leading to an activation of C, i. e. $r_c - 1$ tokens, has to be assumed. Furthermore, it must be assumed that subsequent tokens arrive as early as possible. These assumptions are captured by shifting the upper producer token function upwards $r_c - 1$ tokens. The shifted function shall be called $\bar{\tau}_p^u$.

$$\bar{\tau}_p^u = \tau_p^u + r_c - 1$$

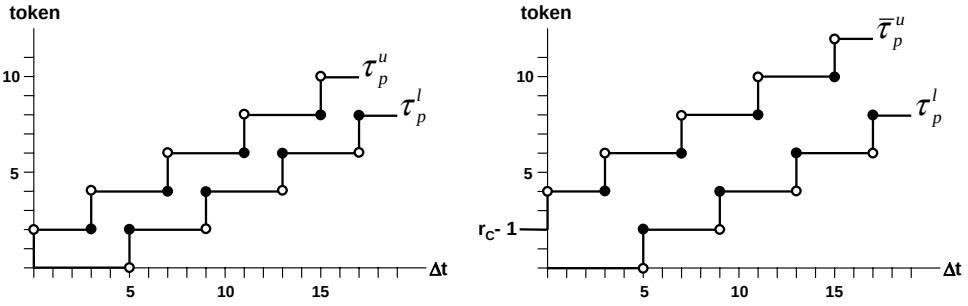
To correctly construct the lower consumer token function, the minimum initial number of tokens in the buffer, i. e. 0, has to be assumed. Furthermore, it must be assumed that subsequent tokens arrive as late as possible. These assumptions are already expressed by the lower producer token function.

Upper and lower consumer token functions can now be constructed. They step as soon as possible, without rising above $\bar{\tau}_p^u$ and τ_p^l, respectively. Otherwise, non-existent tokens might be consumed.

$$\tau_c^u = \left\lfloor \frac{\bar{\tau}_p^u}{r_c} \right\rfloor * r_c \tag{5.5}$$

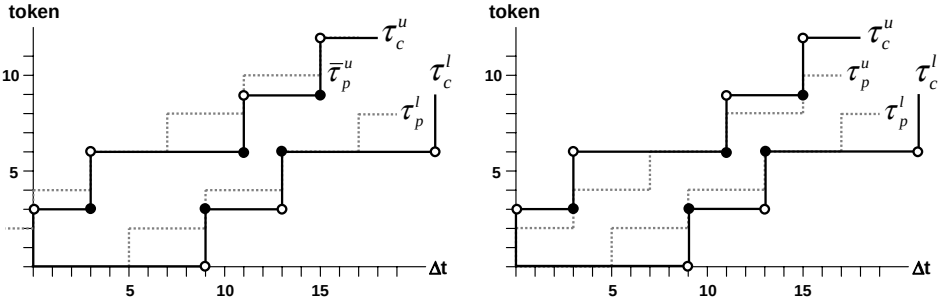
$$\tau_c^l = \left\lfloor \frac{\tau_p^l}{r_c} \right\rfloor * r_c \tag{5.6}$$

In the final step, the consumer token functions are transformed into activating event functions, where each arrival of r_c tokens leads to one activation of task C according to equations 5.3 and 5.4.



(a) original (b) shifted upper producer token function

Figure 5.3. Upper and lower *producer* token functions for our data rate transition example



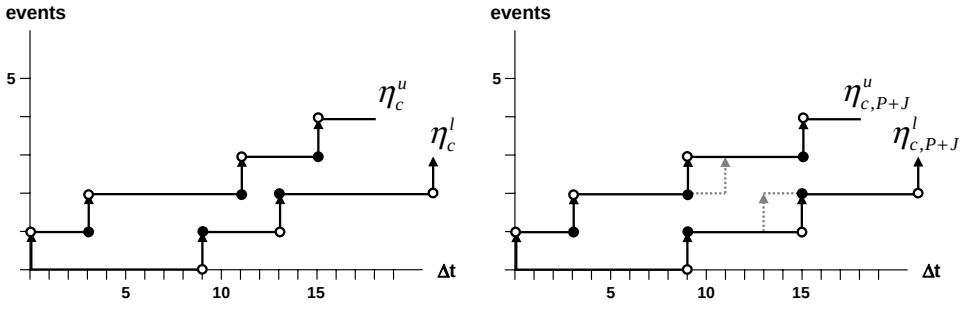
(a) in comparison to $\bar{\tau}_p^u$ and τ_p^l (b) in comparison to τ_p^u and τ_p^l

Figure 5.4. Upper and lower *consumer* token functions for our data rate transition example

Let us apply these results to the example in Fig. 5.2. Let us assume that task P produces 2 tokens per execution with the following *periodic with jitter* output event model:

$$\mathcal{P}_p = 4, \quad \mathcal{J}_p = 1$$

The upper and lower *producer* token functions τ_p^u and τ_p^l are shown in Fig. 5.3(a). In Fig 5.3(b), the upper producer token function has been shifted upwards by $r_c - 1 = 2$. The resulting consumer token functions τ_c^u and τ_c^l are shown in Fig. 5.4(a). As required, they step as soon as



(a) exact

(b) *periodic with jitter* approximation (the dotted lines represent the exact curves from Fig. 5.5(a))

Figure 5.5. Activating event functions for consumer task C in our data rate transition example

possible, without rising above $\bar{\tau}_p^u$ and τ_p^l, respectively. For comparison, $\bar{\tau}_p^u$ and τ_p^l are also shown as dotted lines.

While $\bar{\tau}_p^u$ is helpful for the construction process, it is otherwise not useful to describe event models. It is more interesting to compare the upper consumer token function τ_c^u to the upper producer token function τ_p^u. τ_c^u is at most $r_c - 1$ below $\bar{\tau}_p^u$. Since $\bar{\tau}_p^u$ is exactly $r_c - 1$ above τ_p^u, it follows that τ_c^u is always equal or above τ_p^u. This is shown in Fig. 5.4(b). Therefore, equation 5.5 can also be written as

$$\tau_c^u = \left\lceil \frac{\tau_p^u}{r_c} \right\rceil * r_c \tag{5.7}$$

This leads to a second interpretation of the transformation result. Consumer token functions are a tight conservative approximation of producer token functions. Conservative, since all possible timing of the arrival of tokens allowed by the consumer token functions (represented by the area between the curves) is also allowed by the producer token functions.

Finally, activating event functions are obtained from consumer token functions using equations 5.3 and 5.4. The result for our example is shown in Fig. 5.5(a). We can now also directly specify the relationship

between output event functions and activating event functions.

$$\eta_c^u = \left\lceil \frac{r_p}{r_c} * \eta_p^u \right\rceil \tag{5.8}$$

$$\eta_c^l = \left\lfloor \frac{r_p}{r_c} * \eta_p^l \right\rfloor \tag{5.9}$$

5.1.3 Data Rate Transitions for *periodic with jitter* Event Models

For the reasons stated in section 3.1.1 we are specifically interested in the influence of data rate transitions on *periodic with jitter* event models. I. e. our goal is to calculate the activating event model $(\mathcal{P}_c, \mathcal{J}_c)$ for the consumer task, which we will need for scheduling analysis, from the output event model $(\mathcal{P}_p, \mathcal{J}_p)$ of the producer task.

The output event model in our initial example is in fact *periodic with jitter*. As can be seen in Fig. 5.5(a), the *exact* activating event functions display irregularly spaced steps and thus cannot be described by a *periodic with jitter* event model. The situation is basically the same as for OR-activation 4.2.2 and has the same implications. We need to find conservative approximations for η_c^u and η_c^l that can be described by a *periodic with jitter* event model $(\mathcal{P}_c, \mathcal{J}_c)$. The intended result is shown in Fig. 5.5(b) in comparison to the exact activating event functions (dotted lines).

5.1.3.1 Consumer Activation Period

Calculation of the activating event model period $\mathcal{P}_c$ is straightforward [61].

$$\mathcal{P}_c = \mathcal{P}_p * \frac{r_c}{r_p} \tag{5.10}$$

5.1.3.2 Consumer Activation Jitter

Calculation of the activating event model jitter $\mathcal{J}_c$ is more complex. A conservative approximation of η_c^u and η_c^l with a *periodic with jitter* event model implies that equations 5.8 and 5.9 are replaced by the following inequations.

$$r_c * \left\lceil \frac{\Delta t + \mathcal{J}_c}{\mathcal{P}_c} \right\rceil \geq r_p * \left\lceil \frac{\Delta t + \mathcal{J}_p}{\mathcal{P}_p} \right\rceil \tag{5.11}$$

$$r_c * \max\left(0, \left\lfloor \frac{\Delta t - \mathcal{J}_c}{\mathcal{P}_c} \right\rfloor\right) \leq r_p * \max\left(0, \left\lfloor \frac{\Delta t - \mathcal{J}_p}{\mathcal{P}_p} \right\rfloor\right) \tag{5.12}$$

In order to be as accurate as possible, we are interested in the minimum jitter that satisfies both inequations. Using a similar line of argumentation as for OR-activation (section 4.2.2.2), it can be proven that both inequations are satisfied by the same minimum jitter. In the following, the upper approximation (inequation 5.11) is used. Since the left and right sides of this inequation are only piecewise continuous, the inequation cannot be simply transformed to obtain the desired minimum jitter. The solution used here is to evaluate inequation 5.11 piecewise for each interval $]\Delta t_i, \Delta t_{i+1}]$, during which the right side of the inequation has a constant value $k_i \in \mathbb{N}$. For each constant piece of the right side, a condition for a *local jitter* $\mathcal{J}_{c,i}$ is obtained that satisfies the inequation for all $\Delta t : \Delta t_i < \Delta t \leq \Delta t_{i+1}$. For each constant piece of the right side, inequation 5.11 becomes

$$r_c * \left\lceil \frac{\Delta t + \mathcal{J}_{c,i}}{\mathcal{P}_c} \right\rceil \quad \geq \quad k_i \quad ; \quad \Delta t_i < \Delta t \leq \Delta t_{i+1} \,,\, k_i \in \mathbb{N}$$

Since the left side of this inequation is monotonically increasing with Δt, it is sufficient to evaluate it for the smallest value of Δt, which approaches Δt_i. I. e.

$$\begin{aligned}
& r_c * \lim_{\epsilon \to +0} \left\lceil \frac{\Delta t_i + \epsilon + \mathcal{J}_{c,i}}{\mathcal{P}_c} \right\rceil \geq k_i \quad ; \quad k_i \in \mathbb{N} \\
\Leftrightarrow \quad & \lim_{\epsilon \to +0} \frac{\Delta t_i + \epsilon + \mathcal{J}_{c,i}}{\mathcal{P}_c} > \left\lceil \frac{k_i}{r_c} - 1 \right\rceil \\
\Leftrightarrow \quad & \lim_{\epsilon \to +0} (\mathcal{J}_{c,i} + \epsilon) > \left\lceil \frac{k_i}{r_c} - 1 \right\rceil * \mathcal{P}_c - \Delta t_i \\
\Leftrightarrow \quad & \mathcal{J}_{c,i} \geq \left\lceil \frac{k_i}{r_c} - 1 \right\rceil * \mathcal{P}_c - \Delta t_i \qquad (5.13)
\end{aligned}$$

The global minimum jitter is then the smallest value which satisfies all local jitter conditions. Since the ratio between r_p and r_c is rational, η_c^u displays a pattern of distances between steps which repeats periodically every LCM($\mathcal{P}_p$, $\mathcal{P}_c$) (the macro period). Therefore, as was the case for OR-activation, it is sufficient to perform above calculation for one macro period. An algorithm can be found in [41].

5.1.4 Example

Let us return to our initial example and calculate the activating event model for consumer task C. According to equation 5.10 we obtain the following activation period.

$$\mathcal{P}_c \quad = \quad 4 * \frac{3}{2} = 6$$

Activation jitter is calculated according to inequation 5.11, which becomes

$$3 * \left\lceil \frac{\Delta t + \mathcal{J}_{c,i}}{6} \right\rceil \quad \geq \quad 2 * \left\lceil \frac{\Delta t + 1}{4} \right\rceil \quad ; \quad \Delta t_i < \Delta t \leq \Delta t_{i+1}$$

Continuous pieces of the right side of this inequation now have to be found. The first continuous piece is for $0 < \Delta t \leq 3$. According to inequation 5.12

$$\begin{aligned} & 3 * \left\lceil \frac{\Delta t + \mathcal{J}_{c,0}}{6} \right\rceil \quad \geq \quad 2 \quad ; \quad 0 < \Delta t \leq 3 \\ \Leftrightarrow \qquad & \mathcal{J}_{c,0} \quad \geq \quad 0 * 6 - 0 \; = \; 0 \end{aligned}$$

The following table shows all relevant constant pieces and the resulting local $\mathcal{J}_{c,i}$.

Δt range	k_i	local $\mathcal{J}_{c,i}$
$0 < \Delta t \leq 3$	2	$\mathcal{J}_{c,0} \geq 0 * 6 - 0 = 0$
$3 < \Delta t \leq 7$	4	$\mathcal{J}_{c,1} \geq 1 * 6 - 3 = 3$
$7 < \Delta t \leq 11$	6	$\mathcal{J}_{c,2} \geq 1 * 6 - 7 = -1$
$11 < \Delta t \leq 15$	8	$\mathcal{J}_{c,3} \geq 2 * 6 - 11 = 1$
$15 < \Delta t \leq 19$	10	$\mathcal{J}_{c,4} \geq 3 * 6 - 15 = 3$

Please note that the a lower bound of -1 in the third line is purely hypothetical, since the jitter cannot be less than zero. The last table entry is there only to show that starting with $\Delta t > 3$ the pattern repeats every $\Delta t = \text{LCM}(\mathcal{P}_p, \mathcal{P}_c) = 3 * 4 = 12$ time units.

Finally, we take the largest $\mathcal{J}_{c,i}$ to obtain the activating event model jitter.

$$\mathcal{J}_c = 3$$

5.1.5 Data Rate Transitions for *sporadic with jitter* Input Event Models

If the producer output event model is *sporadic with jitter*, then the same rules to calculate the activating event model apply in the presence of a data rate transition as for *periodic with jitter* event models. The only difference of course is that the activating event model is also *sporadic with jitter*.

5.1.6 Rate Transition Incurred Delay and Backlog

We would like to calculate the best- and worst-case token delay, and the maximum token backlog and consequently the required buffer size

at the consumer task input due to a data rate transition. This buffering is required on top of the execution buffer that holds tokens from the moment the consumer task is activated until execution of the task has been completed (section 3.6.7). We will consider token buffering due to a combination of these effects in section 5.5.

The best-case rate transition delay is obviously zero, since each consumer activation is the result of at least one token arriving from the producer. Consequently, that token does not have to wait for additional tokens. The worst-case delay is incurred for the smallest set of tokens that can remain in the rate-transition buffer after a consumer activation. Let the size of this set be N_{min}. N_{min} can be obtained by starting with the initial number of tokens in the input buffer, and iterating one macro period to obtain each possible number of tokens in the buffer. Without loss of generality, let us assume that the initial number of tokens in the input buffer is 0.

If N_{min} tokens remain in the rate-transition buffer, the consumer is activated after n producer activations, where n is the *smallest* integer solution for the following inequation.

$$n * r_p + N_{min} \geq r_c \quad \Leftrightarrow \quad n \geq \frac{r_c - N_{min}}{r_p}$$

I.e.

$$n \quad = \quad \left\lceil \frac{r_c - N_{min}}{r_p} \right\rceil$$

The maximum activation delay is obtained if the next n groups of r_p tokens arrive as late as possible. Consequently

$$\text{delay} \quad \leq \quad \left\lceil \frac{r_c - N_{min}}{r_p} \right\rceil * \mathcal{P}_p + \mathcal{J}_p \tag{5.14}$$

The maximum backlog due to data rate transitions and hence the required buffer size is

$$\text{backlog} \quad \leq \quad r_c - 1 \tag{5.15}$$

For our example, we obtain

$$\text{delay} \leq 5\,, \quad \text{backlog} \leq 2$$

5.1.7 Special Cases

In this section, we take a brief look at special combinations of producer and consumer data rates.

5.1.7.1 Consumer Data Rate is an Integer Multiple of Producer Data Rate

If the consumer data rate is $n \in \mathbb{N}$ times the producer data rate, then the consumer token curves step periodically every $n * \mathcal{P}_p$ after the initial jitter. If $\mathcal{J}_p = 0$, then $\mathcal{J}_c$ also $= 0$. If $\mathcal{J}_p \neq 0$, then the upper producer token curve is shifted left by $\mathcal{J}_p$ for all $\Delta t > 0$, and the lower producer token curve is shifted right. Consequently, the upper and lower consumer token curve are also shifted left respectively right by $\mathcal{J}_p$. I. e. the jitter calculation in section 5.1.3 can be replaced by the simple equation

$$\mathcal{J}_c \;=\; \mathcal{J}_p \quad ; \qquad \mathcal{P}_c \,=\, n * \mathcal{P}_p \,, \quad n \in \mathbb{N} \tag{5.16}$$

5.1.7.2 Producer Data Rate is an Integer Multiple of Consumer Data Rate

If the producer data rate is $n \in \mathbb{N}$ times the consumer data rate, then each arrival of r_p tokens immediately leads to n consumer activations. I. e. the jitter calculation in section 5.1.3 can be replaced by the simple equation

$$\mathcal{J}_c \;=\; (n-1) * \mathcal{P}_p + \mathcal{J}_p \quad ; \qquad \mathcal{P}_p \,=\, n * \mathcal{P}_c \,, \quad n \in \mathbb{N} \tag{5.17}$$

Rate-transition buffering is not required.

5.1.7.3 Identical Producer and Consumer Data Rates

If $r_p = r_c$, then a tight conservative approximation with step-height r_c of the upper producer token curve is the curve itself, and likewise for lower token curves. I. e. in the absence of a data rate transition, the consumer's activating event model equals the producer's output event model, as would have been expected.

$$\mathcal{J}_c \;=\; \mathcal{J}_p \quad ; \qquad \mathcal{P}_p \,=\, \mathcal{P}_c \tag{5.18}$$

5.2 Data Rate Intervals

Conditional control-flow inside a tasks can lead to the situation that the task consumes or produces a non-constant number of tokens per execution. I. e. producer and consumer data rates become intervals: $[r_{p,min}, r_{p,max}]$ for a producer task and $[r_{c,min}, r_{c,max}]$ for a consumer task. Communication between tasks with data rate intervals can always lead to a data rate transition.

In general, we do not assume any correlation between the number of produced tokens and the number of consumed tokens. A lower consumer data rate of zero is not allowed, since without additional information a

bounded communication buffer cannot be guaranteed. A lower producer data rate of zero corresponds to a *sporadic* output event model.

We would like to extend the results for data rate transitions with fixed data rates (section 5.1) to calculate consumer token functions and activating event functions in the presence of data rate intervals. Before doing so, an interpretation issue has to be addressed regarding the minimum number of tokens required for one activation of consumer task C. Our interpretation is as follows: $r_{c,max}$ tokens are required for one activation of C. This is because the total number of tokens consumed is not known a priory, and we do not want C to stall for lack of tokens. Stalling for the lack of token is problematic for scheduling, since it can lead to deadlocks. This interpretation is consistent with most models of computation. However, the results from this section remain equally valid if only $r_{c,min}$ tokens are required for one activation of C.

Following the approach in section 5.1, to construct the upper consumer token function, we assume the maximum number of initial tokens at the input of C not leading to an activation. We also assume that as many additional tokens as possible arrive as soon as possible. To construct the lower consumer token function, zero initial tokens at the input of C are assumed, and as few additional tokens as possible arriving as late as possible.

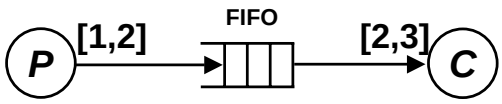


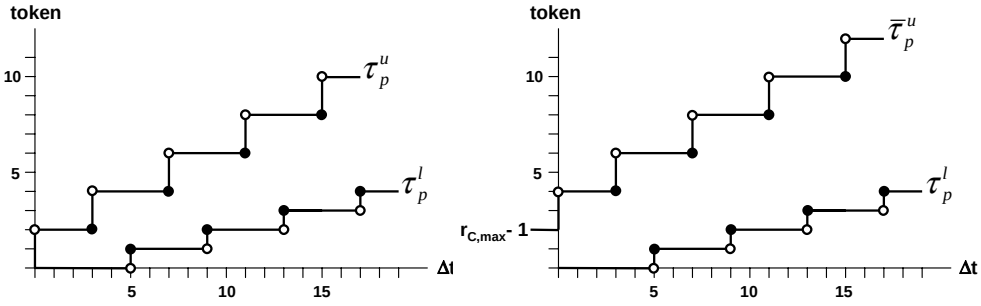
Figure 5.6. Example of a data rate transition with data rate intervals

Let us consider the example in Fig. 5.6 with producer task P and consumer task C, where

$$r_p = [r_{p,min}, r_{p,max}] = [1, 2] \qquad \mathcal{P}_p = 4 \qquad \mathcal{J}_p = 1$$
$$r_c = [r_{c,min}, r_{c,max}] = [2, 3]$$

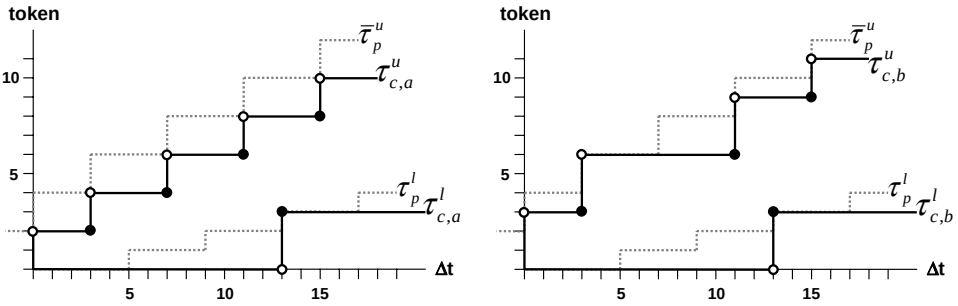
The upper and lower producer token functions are shown in Fig. 5.7(a). In Fig. 5.7(b) the upper producer token function has been shifted upwards by $r_{c,max} - 1$ in analogy to Fig. 5.3(b). Since the consumer data rate is not fixed, there exists no single upper and lower consumer token function. Figs. 5.8 and 5.9 show three possible scenarios[1]. As can

[1]Remember that C requires at least 3 tokens for one activation. Therefore, no two simultaneous activations can be observed when 4 tokens are available, even though it is possible that only 4 tokens are consumed in two subsequent consumer executions.



(a) original (b) shifted upper producer token function

Figure 5.7. Upper and lower *producer* token functions for our data rate transition with intervals example



(a) (b)

Figure 5.8. Two possible sets of upper and lower *consumer* token functions for our data rate transition with intervals example

be seen, depending on the sequence of consumer data rates, a different function can dominate the other functions for a particular Δt.

We could now determine all relevant sequences of consumer data rates and then define the upper consumer token function to be the maximum off all upper 'consumer token scenario functions'. Likewise, we could define the lower consumer token function to be the minimum off all lower 'consumer token scenario functions'. However, in the end we are interested in the upper and lower *activating event functions* for the consumer task. Definitely, it is not possible to obtain a larger number of activations for any Δt than in the scenario in which the minimum number of tokens

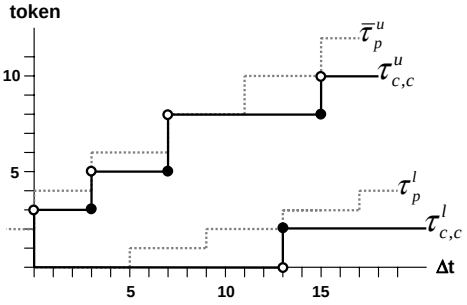


Figure 5.9. A third possible set of upper and lower *consumer* token functions for our data rate transition with intervals example

are consumed during each consumer execution (upper 'consumer token scenario function' in Fig. 5.8(a)). Likewise, it is not possible to obtain a smaller number of activations for any Δt than in the scenario in which the maximum number of tokens are consumed during each consumer execution (lower 'consumer token scenario function' in Fig. 5.8(a)).

We conclude that in the presence of a data rate transition with intervals, the upper activating event function is calculated using the maximum producer data rate $r_{p,max}$, the minimum consumer data rate $r_{c,min}$ and the maximum number of initial tokens at the input of C. To calculate the lower activating event function, the minimum producer data rate $r_{p,min}$, the maximum consumer data rate $r_{c,max}$ and zero initial tokens at the input of C are used. Equations 5.8 and 5.9 are modified accordingly.

$$\eta_c^u = \left\lceil \frac{r_{p,max}}{r_{c,min}} * \eta_p^u \right\rceil \tag{5.19}$$

$$\eta_c^l = \left\lfloor \frac{r_{p,min}}{r_{c,max}} * \eta_p^l \right\rfloor \tag{5.20}$$

5.2.1 Data Rate Transitions for *periodic with jitter* Event Models

Obviously, two sets of period and jitter values are now required for a conservative *periodic with jitter* approximation of both the upper and the lower activating event functions. The situation is similar to OR-concatenation of a combination of *periodic with jitter* and *sporadic with jitter* input event models (section 4.2.4). As was explained there, the alternatives are to either use a single, very conservative *sporadic with jitter* approximation with parameters that correspond to the upper activating event function. The second, tighter alternative is to extend our

system-level analysis framework to propagate two separate event models for each event stream.

For our example, we obtain two *periodic with jitter* event models with the following properties according to equations 5.18 (upper event model) and 5.16 (lower event model).

$$\begin{aligned} \eta^u_{c,\mathcal{P}+\mathcal{J}}: \quad & \mathcal{P}^u_c = 4 \quad & \mathcal{J}^u_c = 1 \\ \eta^l_{c,\mathcal{P}+\mathcal{J}}: \quad & \mathcal{P}^l_c = 12 \quad & \mathcal{J}^l_c = 1 \end{aligned}$$

5.2.2 Data Rate Transitions for *sporadic with jitter* Input Event Models

If the producer output event model is *sporadic with jitter*, then we proceed for the upper activating event model as in the periodic case. The lower activating event function is obviously zero for all Δt, since the lower output event function is zero. Therefore, we do not need two separate *periodic with jitter* event models. Instead, we use the parameters of upper activating event model and additionally set it to *sporadic*.

5.2.3 Rate Interval Transition Incurred Token Delay and Backlog

As in section 5.1, the worst-case delay is incurred for the smallest set of tokens N_{min} that can remain in the rate-transition buffer after a consumer activation. This delay is maximized if subsequently as few tokens as possible arrive as late as possible. Consequently, in correspondence to inequation 5.14

$$\text{delay} \quad \leq \quad \left\lceil \frac{r_{c,max} - N_{min}}{r_{p,min}} \right\rceil * \mathcal{P}_p + \mathcal{J}_p \tag{5.21}$$

In correspondence to inequation 5.15, the possible backlog due to data rate transitions and hence the required buffer size is

$$\text{backlog} \quad \leq \quad r_{c,max} - 1 \tag{5.22}$$

For our example, we obtain

$$\text{delay} \leq 9\,, \quad \text{backlog} \leq 2$$

5.3 Combination with Multiple Inputs

In the previous chapter we showed how to calculate activating functions for a consumer task C with multiple inputs from the individual input event functions. In this chapter we have shown how to calculate activating functions for a consumer task C with a single input in the

presence of a data rate transition between producer task P and consumer task C. In this section, both results are combined.

Let us review the equations that were derived for activating event functions in the previous sections.

AND-activation (equations 4.1 and 4.2).

$$\begin{aligned} \eta^u_{AND} &= \max\{\eta^u_i\} \\ \eta^l_{AND} &= \min\{\eta^l_i\} \end{aligned}$$

OR-activation (equations 4.11 and 4.12).

$$\begin{aligned} \eta^u_{OR} &= \sum_{i=1}^{n} \eta^u_i \\ \eta^l_{OR} &= \sum_{i=1}^{n} \eta^l_i \end{aligned}$$

Data rate transitions with fixed data rates (equations 5.8 and 5.9).

$$\begin{aligned} \eta^u_c &= \left\lceil \frac{r_p}{r_c} * \eta^u_p \right\rceil \\ \eta^l_c &= \left\lfloor \frac{r_p}{r_c} * \eta^l_p \right\rfloor \end{aligned}$$

Data rate transitions with data rate intervals (equations 5.19 and 5.20).

$$\begin{aligned} \eta^u_c &= \left\lceil \frac{r_{p,max}}{r_{c,min}} * \eta^u_p \right\rceil \\ \eta^l_c &= \left\lfloor \frac{r_{p,min}}{r_{c,max}} * \eta^l_p \right\rfloor \end{aligned}$$

A combination of AND- and OR-activation has been discussed in section 4.3 and is not considered here. The combination of AND-activation with data-rate intervals is not acceptable, since bounded buffers cannot be guaranteed at all task inputs. The remaining combinations can be easily obtained by first performing rate-transition calculation, and then multiple-input calculation.

AND-activation combined with fixed data rate transitions.

$$\eta^u_{AND} = \max \left\lceil \frac{r_{p,i}}{r_{c,i}} * \eta^u_{p,i} \right\rceil \tag{5.23}$$

$$\eta^l_{AND} = \min \left\lfloor \frac{r_{p,i}}{r_{c,i}} * \eta^l_{p,i} \right\rfloor \tag{5.24}$$

OR-activation combined with fixed data rate transitions.

$$\eta^u_{OR} = \sum_{i=1}^{n} \left\lceil \frac{r_{p,i}}{r_{c,i}} * \eta^u_{p,i} \right\rceil \tag{5.25}$$

$$\eta^l_{OR} = \sum_{i=1}^{n} \left\lfloor \frac{r_{p,i}}{r_{c,i}} * \eta^l_{p,i} \right\rfloor \tag{5.26}$$

OR-activation combined with interval data rate transitions.

$$\eta^u_{OR} = \sum_{i=1}^{n} \left\lceil \frac{r_{p,max,i}}{r_{c,min,i}} * \eta^u_{p,i} \right\rceil \tag{5.27}$$

$$\eta^l_{OR} = \sum_{i=1}^{n} \left\lfloor \frac{r_{p,min,i}}{r_{c,max,i}} * \eta^l_{p,i} \right\rfloor \tag{5.28}$$

The application to *periodic with jitter* event models and *sporadic with jitter* event models is straight-forward using the results from sections 4.1.3, 4.2.2, 5.1.3 and 5.2.1.

5.4 Chaining of Rate Transitions, Multiple Inputs and EAFs

An interesting question is in which order to chain rate transitions, multiple inputs and EAFs when calculating activating event functions of a consumer task. In the previous section we already established that rate transition calculations are performed prior to multiple input calculations. However, EAFs are a more complicated case.

The purpose of an EAF is to change the possible activation timing of a task in order to meet some requirement (section 3.6.7). In the presence of a rate transition this implies that it makes little sense to imagine that an EAF changes the output event model of the producer task, and to then calculate the input event model for the consumer, since a rate transition changes the event model. For example, if strictly periodic consumer activation was required, then this could not be enforced in general by a periodic-EAF followed by a rate transition, due to the jitter introduced by the rate transition. We conclude that EAF-calculation needs to follow

rate transition calculation. This is consistent with our model that an EAF delays task activation after an input event (section 3.6.7).

A similar argument applies for a combination of OR-activation and EAFs. Assume an OR-activated task with *periodic with jitter* $\neq 0$ input event models with different periods, which requires strictly periodic activation. We cannot satisfy our requirement with individual EAFs at each port that change each input event model into strictly periodic, since the activating event model will have a jitter $\neq 0$ due to the different input event model periods (section 4.2). We therefore conclude that it makes more sense to let EAF-calculation follow OR-concatenation. This implies that a single EAF needs to be specified for the OR-activated task which directly influences activation timing, instead of individual EAFs for each input, where the influence on timing is indirect and not sufficiently controllable. In the next section we will see a straight-forward implementation of such a task-level EAF. Additional EAFs at individual inputs are not forbidden, but serve little purpose.

The case is less clear for a combination of AND-activation and EAFs. If we want to reduce the activation jitter of an AND-activated task, then we can either use a single task-level EAF in analogy to OR-activated tasks, or we can use individual EAFs at those inputs where the input jitter exceeds the required jitter (the activation jitter equals the largest input jitter, section 4.1). The advantage of a single EAF is simplicity and consistency with OR-concatenation. The approach is also applicable if a task is activated by a combination of AND- and OR-concatenations (section 4.3).

In summary, we propose the following order to calculate an activating event models:
rate transition → multiple inputs (AND, OR, combination) → EAF.

5.5 Combined Token Buffering

AND-activation (section 4.1.2) and data rate transitions (section 5.1.6) both require token buffering. This buffering happens on top of activation buffering and EAF buffering (section 3.6.7).

We shall assume that input tokens are buffered separately at each consumer task input. In correspondence to section 3.6.7, tokens are kept in the buffers from the moment of arrival until the instance of the task that was activated by the tokens has been completed. For ease of illustration, the input buffers have been laid out in Fig. 5.10 such that the filling of one column at one input leads to one input event at that input. We assume that the buffers are written and read in the order indicated by the arrows.

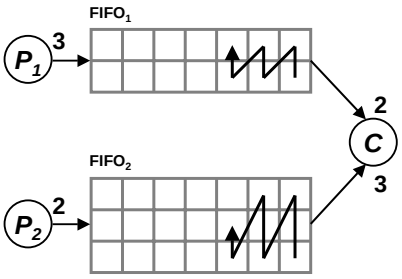
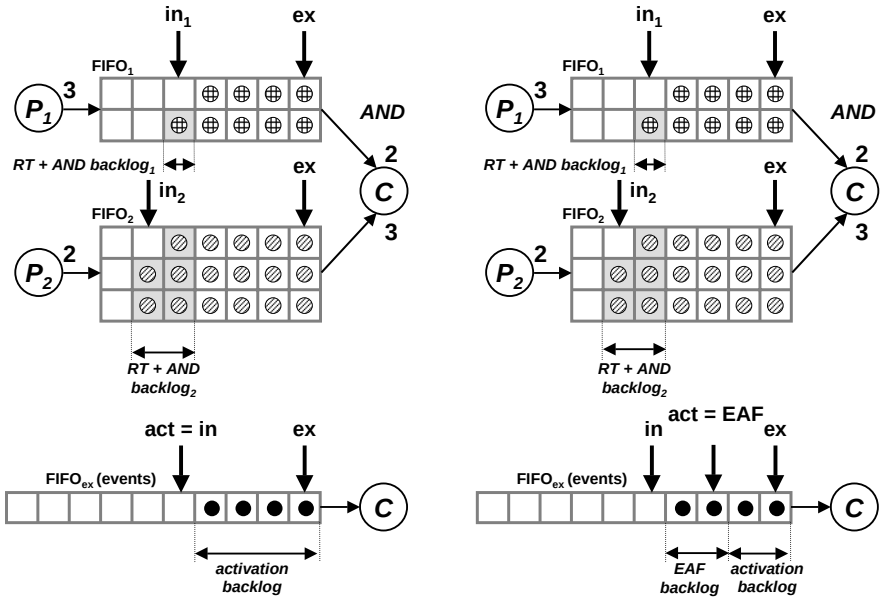


Figure 5.10. Communication buffer (implemented as circular buffer)

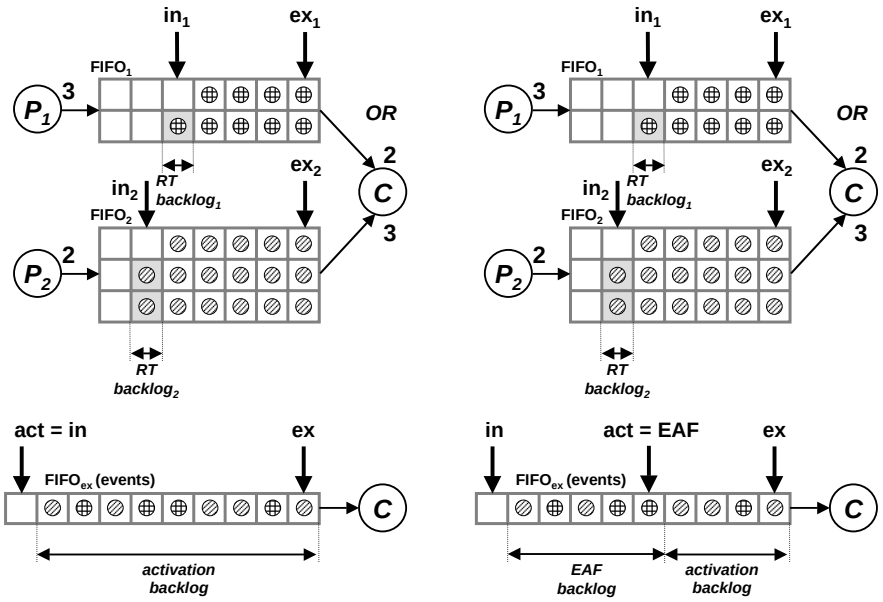


(a) Rate transitions with AND-activation

(b) Rate transitions with AND-activation and EAF

Figure 5.11. Token buffering and activation mechanism for a combination of rate transitions with AND-activation and an optional EAF

A possible solution for a combination of data rate transitions with AND-activation and an optional EAF is shown in Fig. 5.11. The same solution is applicable for a combination of data rate transitions with OR-activation and an optional EAF (Fig. 5.12). Each input buffer has its own rate transition and, in case of AND-activation, AND-concatenation backlog. There is an additional buffer (bottom) which keeps track of



(a) Rate transitions with OR-activation

(b) Rate transitions with OR-activation and EAF

Figure 5.12. Token buffering and activation mechanism for a combination of rate transitions with OR-activation and an optional EAF

the number, and in case of OR-activation, order in which input events occurred at the two inputs. This buffer is also used to implement an optional task-level EAF (Figs. 5.11(b) and 5.12(b)) in accordance with section 3.6.6.

5.6 Summary and Conclusion

In this chapter we looked at tasks that consume or produce multiple tokens per execution. We showed how to calculate activating event functions for a consumer task in the presence of data rate transitions. We also calculated required communication buffers and incurred token delay due to a data rate transition. We considered both data rate transitions between fixed data rates, and between data rate intervals. In both cases, we had to conservatively approximate the exact activating event functions for the consumer task, in order to obtain period and jitter parameters required in our compositional performance analysis approach.

We then combined those results with the results from the previous chapter in order to calculate activating event models and the required

buffers in the presence of both data rate transitions and multiple activating inputs. We showed how input buffering can be implemented that also allows traffic shaping using EAFs. This allows us to analyze the performance of complex feed-forward applications. In the next chapter, this work is extended to applications with cyclic task dependencies.

Chapter 6

CYCLIC TASK DEPENDENCIES

Tasks with multiple inputs allow to build cyclic dependencies. A typical application is a control loop, where one task represents the controller and the other task a model of the controlled system. A task graph with a cycle is shown in Fig. 6.1.

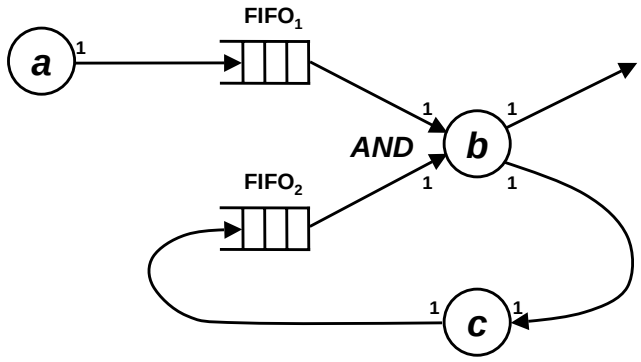


Figure 6.1. Example of a task graph with a cycle.

In this chapter, we focus on single-rate cycles, i. e. without data rate transitions, and with unconditional communication. Cycles with data rate transitions will be briefly considered at the end of the chapter. We will also consider a special case of cycles with date rate transitions when discussing system-level performance analysis for Simulink designs (chapter 9).

6.1 Single-Rate Cycles

All tasks in a single-rate cycle consume and produce the same fixed number of tokens per execution at their cycle-internal inputs and outputs. This implies that tasks with multiple inputs in a single-rate cycle have to be AND-activated. OR-activated tasks are not allowed, since the output period of an OR-activated task is smaller than the smallest input period (section 4.2). This output period would be eventually propagated around the cycle, leading to yet a smaller output period. A fix-point cannot be reached, and the cycle thus cannot be scheduled[1]. Without loss of generality, only unit-rate cycles are considered. The extension to non-unit single-rates is trivial. The example in Fig. 6.1 is in fact a system with a unit-rate cycle.

Let us further assume that all AND-activated tasks in a unit-rate cycle have exactly one cycle-internal input. Then, at least one initial token has to be present at the input of at least one task in the cycle to avoid deadlock [55]. More than one initial token allow to execute cycle tasks in parallel, if they are mapped onto different resources or can be pipelined. The number of tokens in the cycle remains constant, since for every consumed token, one token is produced (atomic buffer update, section 3.6.1). For more complex topologies, SDF scheduling rules [62] allow to calculate the required number and positions of initial tokens to avoid deadlock.

For $\lim_{\Delta t \to \infty}$, all tasks in a unit-rate cycle are activated the same number of times. Consequently, for $\lim_{\Delta t \to \infty}$ the same number of input events must occur at cycle-external inputs of all AND-activated tasks belonging to the same cycle to ensure that external buffers can be bounded. For our compositional performance analysis approach this implies that a cycle must be activated externally with *periodic with jitter* event models, and that all external periods must be the same. An important exception is a cycle with only one external input. In this case, any external event model, in particular *sporadic with jitter* is acceptable since the number of cycle-internal tokens is fixed an thus cannot grow if no external tokens arrive.

If we want to use our performance analysis in the presence of single-rate cycles, we first have to solve a starting-point problem. The input event model at the cycle-internal input of an AND-activated task ultimately depends on the task's activating event model, which cannot be determined without all input event models. A conservative starting

[1] If we allow a combination of OR- and AND-activation (section 4.3), then cycle-external inputs of a multi-input task can be OR-concatenated before AND-concatenation with the cycle internal inputs.

point is to initially assume zero jitter at the cycle-internal input (section 3.6.4). The activating event model of the AND-activated task then equals the external input event model (section 4.1). We can now iterate analysis and event model propagation around the cycle, hoping to find a fix-point.

However, if only one task along the cycle has a response time which is an interval, then after the first round of analysis and event model propagation the internal input jitter of the AND-activated task will be larger than the external input jitter. In our compositional performance analysis approach, this larger jitter will be propagated around the cycle again, resulting in an even larger jitter at the cycle-internal input of the AND-activated task (section 3.5). It is obvious that the jitter appears unbounded if calculated this way.

Non-functional cycles that do not converge by themselves can be made to converge by actively reducing event model jitter through an EAF (section 3.6.6). Unfortunately, such an approach cannot in general be applied to functional cycles, since the additional EAF delay may yield a cycle that cannot be executed fast enough. For example, let us assume an external event model ($\mathcal{P}_{b,ext} = 100$, $\mathcal{J}_{b,ext} = 0$) in Fig. 6.1, and a single initial token in the cycle. If tasks b and c have response time intervals of $[10, 12]$ and $[20, 25]$, respectively, then the time around the cycle is $[30, 37]$. If on the other hand a periodic resynchronization buffer is inserted between tasks c and b in order to reduce the jitter to zero, then according to [96] the time around the cycle increases to $[130, 244]$, thus exceeding the external period.

The problem boils down to the fact that event model propagation as presented so far captures neither correlations between the timing of events in different event streams, nor the fact that the number of tokens in a cycle is fixed. Therefore, the activation jitter for the AND-activated task is calculated very conservatively.

6.2 Analysis Idea

In order to enable performance analysis in the presence of cyclic task dependencies, we need to consider event correlations arising from functional cycles. It would be especially useful if we could show for a particular cycle that the cycle-internal input of the AND-concatenated task cannot increase the activation jitter of that task. This would allow us to ignore the cycle-internal input of the AND-concatenated task during response-time calculation, effectively cutting the cycle and yielding a purely feed-forward system. This feed-forward system could then be analyzed using our compositional performance analysis approach without further extensions, and without the need to propagate event models

around the cycle multiple times in search of a fix-point, which as shown above fails.

The interesting part is to determine conditions under which this approach is valid. Let us for now consider unit-rate cycles with exactly one AND-concatenated task with one cycle-external and one cycle-internal input (as in Fig. 6.1). Let us assume an external *periodic with jitter* event model with period $\mathcal{P}_{ext}$ and jitter $\mathcal{J}_{ext}$. Let us define t_{ff}^{min} and t_{ff}^{max} to be the minimum respectively maximum sum of worst-case response times of all tasks belonging to a cycle (the 'time around the cycle') as obtained through analysis of the corresponding feed-forward system[2]. Furthermore, let us assume for now that all tokens in the cycle are initially at the cycle-internal input of the AND-concatenated task. The number of initial tokens in the cycle shall be $M \geq 1 \in \mathbb{N}$.

At system startup, the first M tokens arriving at the cycle-external input will immediately activate the AND-concatenated task together with the M tokens already waiting at the cycle-internal input. We assume that the external input event model is valid from the moment that the first external token arrives [96]. Consequently, the activating event model equals the external input event model for the first M activations. Let us now consider the following activations.

6.3 Cycles with one initial token

Consider a cycle where after analysis of the corresponding feed-forward system, $t_{ff}^{max} \leq \mathcal{P}_{ext}$. Let us assume that exactly one initial token is available in the cycle, and that the AND-concatenated task (task b in Fig. 6.1) has just been activated for the first time. Consequently, the buffer at the cycle-internal input of task b is now empty (since there is only one token in the cycle), and thus no further activation of task b is possible at that moment. It will take between t_{ff}^{min} and t_{ff}^{max} time units until the next token becomes available at the cycle-internal input of task b.

The first external token does not have to wait for an internal token, because the initial token has been placed at the cycle-internal input of the AND-concatenated task (section 6.2). The maximum distance between two consecutive external tokens is $\delta_{ext}^{max}(2) = \mathcal{P}_{ext} + \mathcal{J}_{ext}$ (equation 3.4). From $t_{ff}^{max} \leq \mathcal{P}_{ext}$ follows that it is not possible that the $2nd$ external token arriving as *late* as possible after the $1st$ external token has to wait for an internal token.

[2]This of course implies that the feed-forward system in not over-loaded and can be analyzed.

The $3rd$ external token can arrive at most $\delta_{ext}^{max}(3) = 2 * \mathcal{P}_{ext} + \mathcal{J}_{ext}$ after the $1st$ external token. Therefore, if both the $2nd$ and the $3rd$ external tokens arrive as late as possible, then the $3rd$ arrives $\mathcal{P}_{ext}$ after the $2nd$. From $t_{ff}^{max} \leq \mathcal{P}_{ext}$ follows that the $3rd$ external token arriving as *late* as possible after the $1st$ external token cannot wait for an internal token, even if the $2nd$ external token also arrived as *late* as possible. This argument can be extended to all further tokens. We infer that no external token arriving as late as possible has to wait for an internal token.

Activation of task b also cannot happen earlier than the arrival of an external token. Therefore, the activating event model of task b is conservatively captured by the external input event model. We conclude that our approach is valid for a cycle with $M = 1$ initial token, for which $t_{ff}^{max} \leq \mathcal{P}_{ext}$.

$$\mathcal{P}_{act} = \mathcal{P}_{ext} \quad ; \quad \mathcal{J}_{act} = \mathcal{J}_{ext} \tag{6.1}$$

For example, let us assume that in our system in Fig. 6.1 task b is activated externally with ($\mathcal{P}_{b,ext} = 4$, $\mathcal{J}_{b,ext} = 3$). Let us further assume that feed-forward analysis has determined the time around the cycle to be $[t_{ff}^{min}, t_{ff}^{max}] = [2, 3]$. I. e. each internal input event follows between $[2, 3]$ time units after the previous activating event. Fig. 6.2 shows a snapshot of a sequence of external, internal and activating events for task b (numbers indicate corresponding input events and the resulting activating event). The first internal event is due to the initial token. As can be seen, activating event timing can be described by the same event model as external input event timing. If on the other hand analysis of the corresponding feed-forward system determines $t_{ff}^{max} > \mathcal{P}_{ext}$, then this statement is no longer true, since for example the $3rd$ internal event could occur later than the latest possible $3rd$ external event.

In Fig. 6.2 it can also be seen that an *early* external token may have to wait for an internal token since two token arrivals at the cycle-internal input of task b cannot follow closer than t_{ff}^{min}, and thus

$$\delta_{act}^{min}(2) \quad = \quad \begin{cases} \delta_{ext}^{min}(2) & ; \quad t_{ff}^{min} \leq \delta_{ext}^{min}(2) \\ t_{ff}^{min} & ; \quad t_{ff}^{min} > \delta_{ext}^{min}(2) \end{cases} \tag{6.2}$$

Effectively, if $t_{ff}^{min} > \delta_{ext}^{min}(2)$, then the cycle acts like a d_{min}-EAF with $d_{min} = t_{ff}^{min}$ (section 3.6.6). This additional effect of the cycle does not *require* a new scheduling analysis, since the possible activation timing is only tightened. All possible event timing in the tighter model is already included in the wider model. Therefore, the results in equation 6.1 remain valid. However, if an extended *periodic with jitter* event model

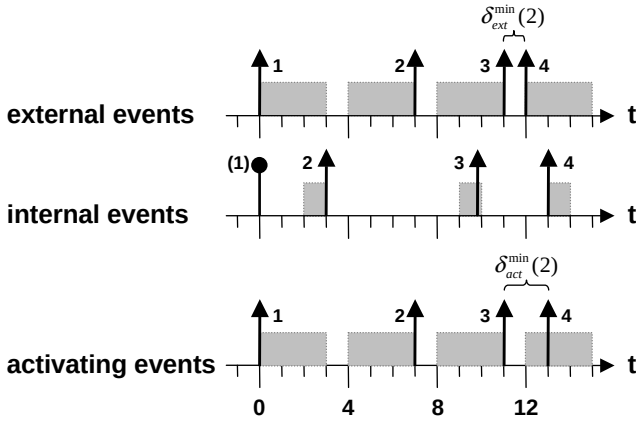


Figure 6.2. Possible event sequence for our cycle example. Gray boxes indicate jitter intervals during which an event can occur. Note that line 2 displays the possible timing of internal events depending on the previous activating event, while lines 1 and 3 display the possible timing of events independent of previous events.

with a $d_{min} = \delta^{min}(2)$ parameter is used as is the case in our compositional performance analysis approach, then it is worthwhile to perform scheduling analysis again with the tighter activating event model for the AND-concatenated task.

6.3.1 Buffer Calculation

The required buffer sizes inside the cycle are trivial: each buffer requires size one, since exactly one token is circling the cycle. The worst-case AND-concatenation delay at the cycle-internal input of the AND-concatenated task equals $\delta_{ext}^{max}(2) - t_{ff}^{min}$. This value is of no further interest.

Buffer calculation is more interesting at the cycle-external input of the AND-concatenated task. If $t_{ff}^{max} \leq \delta_{ext}^{min}(2)$ then a cycle-external token can never wait for the cycle-internal token, and consequently no AND-buffer is required at the cycle-external input (activation buffer calculation is not affected). However, if $t_{ff}^{max} > \delta_{ext}^{min}(2)$, then even though the activation jitter of the AND-concatenated task is not affected by the cycle-internal input, the required size of the cycle-external buffer may be. For example, consider a cycle-external event model ($\mathcal{P}_{ext} = 6$, $\mathcal{J}_{ext} = 18$) with a minimum distance $\delta_{ext}^{min}(2) = 1$ between events. Let us assume that the worst-case response time of the AND-concatenated task is $r_{AND}^{max} = 1$, and that the worst-case time around the cycle is $t_{ff}^{max} = 5$. If the cycle-internal input is ignored during buffer calculation, then feed-forward analysis will determine that the required external buffer size

for the AND-concatenated task is 1, since $\delta_{ext}^{min}(2) \geq r_{AND}^{max}$, i. e. external events cannot follow closer than the worst case response time of the AND-concatenated task. This result is obviously wrong, since in reality external tokens can backlog while a token is circling the cycle.

A conservative worst-case backlog and delay at the cycle-external input due to AND-concatenation is obtained if it is assumed that external tokens arrive as soon as possible, and that each iteration of the cycle takes as long as possible, i. e. t_{ff}^{max} time units[3]. The results can be obtained by treating the cycle like a d_{min}-EAF with $d_{min} = t_{ff}^{max}$ (section 3.6.6). For the example in this section, this results in an external AND-buffer size of 3 and a worst-case external backlog of 14.

6.4 Cycles with two or more initial tokens

Now consider a cycle where after analysis of the corresponding feed-forward system, $(M-1) * \mathcal{P}_{ext} < t_{ff}^{max} \leq M * \mathcal{P}_{ext}\,;\, M \in \mathbb{N}$. Let us assume that exactly M tokens are available in the cycle, and that the AND-concatenated task has just been activated for the first time. It will take the cycle-internal token that has just been consumed between t_{ff}^{min} and t_{ff}^{max} time units to return to the cycle-internal input of task b, where it becomes the $M+1st$ internal token from the perspective of that task.

The first M external tokens do not have to wait for internal tokens, because the M initial tokens have been placed at the cycle-internal input of the AND-concatenated task (section 6.2). The maximum distance between $M+1$ consecutive external tokens is $\delta_{ext}^{max}(M+1) = M * \mathcal{P}_{ext} + \mathcal{J}_{ext}$ (equation 3.4). From $t_{ff}^{max} \leq M * \mathcal{P}_{ext}$ follows that it is not possible that the $M+1st$ external token arriving as *late* as possible after the $1st$ external token has to wait for the $M+1st$ internal token.

The $M+2nd$ external token can arrive at most $\delta_{ext}^{max}(M+2) = (M+1) * \mathcal{P}_{ext} + \mathcal{J}_{ext}$ after the $1st$ external token. Therefore, if both the $M+2nd$ and the $2nd$ external tokens arrive as late as possible, then the $M+2nd$ arrives $M * \mathcal{P}_{ext}$ after the $2nd$. From $t_{ff}^{max} \leq M * \mathcal{P}_{ext}$ follows that the $M+2nd$ external token arriving as *late* as possible after the $1st$ external token cannot wait for an internal token, even if the $2nd$ external token also arrived as *late* as possible. This argument can be extended to all further tokens. We infer that no external token arriving as late as possible has to wait for an internal token.

[3]Tighter results could be obtained if worst-case times around the cycle would be calculated for individual iterations in a sequence of iterations. This would require a considerable analysis extension and is not further considered in this thesis.

Activation of task b also cannot happen earlier than the arrival of an external token. Therefore, the activating event model of task b is conservatively captured by the external input event model. We conclude that our approach is valid for a cycle with $M > 1$ initial tokens, for which $(M-1) * \mathcal{P}_{ext} < t_{ff}^{max} \leq M * \mathcal{P}_{ext}$.

$$\mathcal{P}_{act} = \mathcal{P}_{ext} \quad ; \quad \mathcal{J}_{act} = \mathcal{J}_{ext}$$

Additionally, we may be able to tighten the minimum distance between $(M+1)$ activating events compared to the minimum distance between $(M+1)$ external input events.

$$\delta_{act}^{min}(M+1) = \begin{cases} \delta_{ext}^{min}(M+1) & ; \quad t_{ff}^{min} \leq \delta_{ext}^{min}(M+1) \\ t_{ff}^{min} & ; \quad t_{ff}^{min} > \delta_{ext}^{min}(M+1) \end{cases} \tag{6.3}$$

Effectively, if $t_{ff}^{min} > \delta_{ext}^{min}(M+1)$, then the cycle acts like a d_{min}-EAF [96] with $d_{min,M+1} = t_{ff}^{min}$.

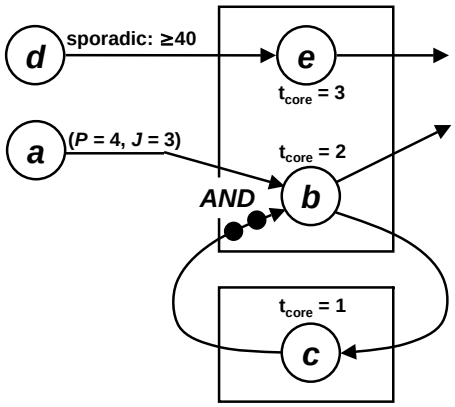


Figure 6.3. Cycle example with 2 initial tokens at the cycle-internal input of AND-concatenated task b. Unit data rates and FIFOs have been omitted.

Consider the example in Fig. 6.3. Task b is activated externally *periodically with jitter* with ($\mathcal{P}_{b,ext} = 4$, $\mathcal{J}_{b,ext} = 3$). It shares a resource with task e which is activated sporadically at most every 40 time units. We assume static priority preemptive scheduling and a higher priority for task e. Task c is mapped onto a separate resource. The core execution times are 3, 2, and 1 for tasks e, b and c, respectively. For this system, feed-forward analysis determines the time around the cycle to be $[t_{ff}^{min}, t_{ff}^{max}] = [3, 7]$.

Fig. 6.4 shows a snapshot of a sequence of external, internal and activating events for task b (numbers indicate corresponding input events

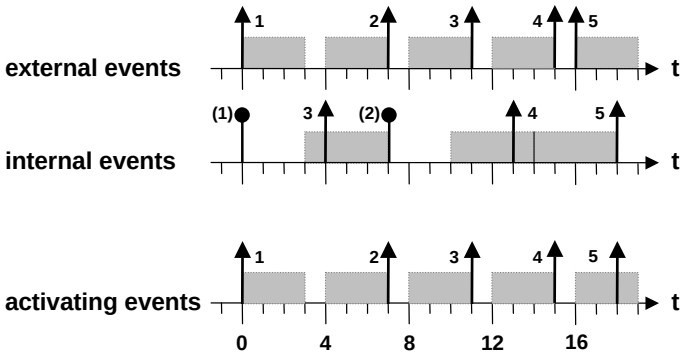


Figure 6.4. Possible event sequence for the cycle example in Fig. 6.3. Gray boxes indicate jitter intervals during which an event can occur. Note that line 2 displays the possible timing of internal events depending on the previous activating event, while lines 1 and 3 display the possible timing of events independent of previous events.

and the resulting activating event). The first two internal events are due to the initial tokens (the $3rd$ internal event can occur before the $2nd$ initial token has led to an activation). As can be seen, activating event timing can be described by the same event model as external input event timing. If we increase core execution times such that analysis of the corresponding feed-forward system determines $t_{ff}^{max} > 2 * \mathcal{P}_{ext}$, then this statement is no longer true, since for example the $4th$ internal event could occur later than the latest possible $4th$ external event.

6.4.1 Buffer Calculation

In the worst-case, the required buffers inside the cycle have size M, since exactly M tokens are circling the cycle. The buffer sizes can be smaller for some cycle tasks, if it can be shown that a particular buffer cannot contain all tokens at the same time. Such an analysis extension is not further considered in this thesis.

If $t_{ff}^{max} \leq \delta_{ext}^{min}(M+1)$ then a cycle-external token can never wait for a cycle-internal token, and consequently no AND-buffer is required at the cycle-external input of task b (activation buffer calculation is not affected). If $t_{ff}^{max} > \delta_{ext}^{min}(M+1)$, then a modification of the approach in section 6.3.1 can be applied using a d_{min}-EAF [96] with $d_{min,M+1} = t_{ff}^{max}$.

6.5 Analyzability Condition

The results obtained in this section can be summarized in the following analyzability condition.

LEMMA 6.1 *A single-rate cycle with one AND-concatenated task with one external input and external period $\mathcal{P}_{ext}$ and one internal input can be treated as a feed-forward system, if the sum of worst-case response times of* all *cycle tasks calculated for the feed-forward system, divided by the number M of initial tokens in the cycle, does not exceed the external period $\mathcal{P}_{ext}$, and if the M tokens are initially at the cycle-internal input of the AND-concatenated task.*

PROOF 6.1 Proof by induction follows from the argumentation in sections 6.3 and 6.4. □

6.6 Cycles with more available than required initial tokens

If the number of initial tokens in the cycle is larger than required by lemma 6.1, then the additional tokens obviously cannot increase the time between arrivals of tokens at cycle-internal inputs of the AND-activated task. Consequently, the cycle-internal input still cannot increase the activation jitter of the AND-concatenated task, and analyzing the corresponding feed-forward system remains a valid approach.

However, the additional tokens can decrease the minimum AND-delay for early external token. A detailed calculation is left for future work. We stay on the safe side by assuming that $\delta_{act}^{min}(M+1) = \delta_{ext}^{min}(M+1)$.

6.7 Cycles with fewer available than required initial tokens

If the number of initial tokens in the cycle is smaller than required by lemma 6.1, then analyzing the corresponding feed-forward system is no longer a valid approach. The cycle may still be schedulable, but then possible phases between events have to be modeled during analysis for several iterations of the cycle. This is a major extension of our compositional performance analysis approach which is beyond the scope of this thesis. Inter-contexts (section 8.2) and the ideas for performance analysis of Simulink (chapter 9) are first steps in this direction.

A much simpler solution is to alert the designer of the problem and indicate how many initial tokens are needed in the cycle to render analysis of the corresponding feed-forward system a valid approach. The designer then has two choices: he can either change the system implementation until lemma 6.1 is satisfied, or increase the number of initial tokens in the cycle. However, the second choice changes properties of the algorithm being implemented, e. g. the dynamics of a control loop, and thus may not always be applicable.

6.8 Self-Cycles

Self-cycles with N initial tokens are an explicit way to indicate that M executions of a task must be completed before the $M + N$th execution is allowed to start. Most single-resource schedulers will only start executing a task again after the previous execution has been completed. Therefore, tasks with self-cycles with more than one initial token would have to be mapped onto multiple resources or pipelined to exploit parallelism. This is not considered further in this thesis.

6.9 System Startup

So far we have assumed that all cycle tokens are initially at the cycle-internal input of the AND-concatenated task. If the initial tokens are placed at any other cycle task input, then it is possible that the first activation is delayed after the arrival of the first external token, and that the next few activating events occur in quicker succession than allowed by the calculated event model. Let us assume that in the system in Fig. 6.1 the initial token is placed at the input of task c, and that the response time interval of task c is $[1, 2]$. As can be seen in Fig. 6.5, if external tokens arrive as soon as possible, and the first internal event arrives as late as possible, then the first 2 activating events violate the ($\mathcal{P}_{b,act} = 4$, $\mathcal{J}_{b,act} = 3$) event model that we obtained in section 6.3. For all subsequent activating events, the results from section 6.3 are valid.

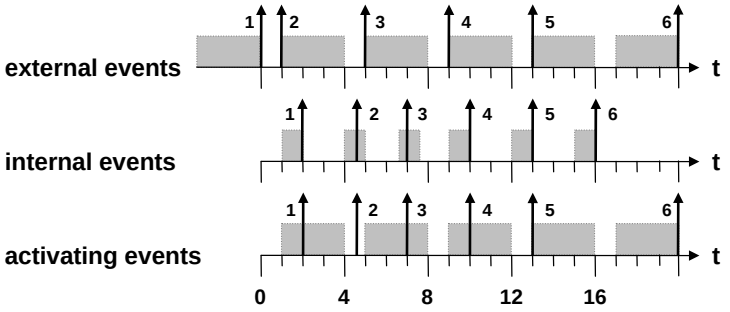


Figure 6.5. Possible event timing at system startup that violates the calculated activating event model. Note the absolute times on the t-axis.

System startup is a problem independent of the presence of cycles, since none of the calculated event models anywhere in the system is valid as long as no events have occurred. This implies that none of the calculated performance numbers is initially valid. For example, a best-case task response time may be initially shorter than calculated, because a second task that would otherwise have certainly led to an

interrupt has not been activated yet. The system needs some time to settle into a consistent state. One idea to control the process of settling in is to build systems in a way that all event models are enforced from the beginning. For example, an operating system could generate dummy task activations as long as no real activating event has occurred.

This idea could also be applied to cycles if the cycle tokens initially were not at the cycle-internal input of the AND-concatenated task. Task activation would consider only the cycle-external input, and read dummy tokens at the cycle-internal input as long as no real token has arrived. However, a thorough investigation of this idea is beyond the scope of this thesis. As long as it has not been performed, the condition in lemma 6.1 restricts the kind of cycles that can be analyzed.

6.10 Nested Cycles

So far, we have considered single cycles. The analysis approach that we derived is obviously also applicable to multiple independent cycles in a system. We now show that it can also be applied to nested cycles, which occur in complex control-loops and filters.

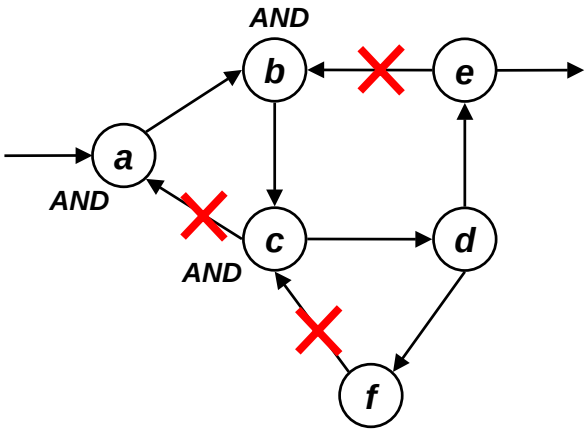


Figure 6.6. Example of a task graph with nested cycles. Unit data rates and FIFOs have been omitted. Edges that need to be cut for corresponding feed-forward analysis are indicated.

Consider the example in Fig. 6.6, which consists of three nested cycles joined at AND-concatenated tasks. Our analysis approach that we developed in this section requires cycles with one external input. This requirement is satisfied by cycle $c \rightarrow d \rightarrow f$ (the innermost cycle). If we determine during performance analysis that this cycle satisfies the conditions in lemma 6.1, then the activating event model of task c does

not depend on the output event model of task f. We can therefore assume that edge $f \rightarrow c$ is cut (indicated by a large 'X' in Fig. 6.6). Consequently, we can now also treat cycle $b \rightarrow c \rightarrow d \rightarrow e$ as an innermost cycle (since the timing of c is not affected by f). If we determine during performance analysis that this cycle also satisfies the conditions in lemma 6.1, then it is correct to assume that edge $e \rightarrow b$ is cut, and cycle $a \rightarrow b \rightarrow c$ also becomes an innermost cycle. If we determine during performance analysis that this cycle also satisfies the conditions in lemma 6.1, then it is also correct to assume that edge $c \rightarrow a$ is cut. The result is a valid corresponding feed-forward system for our system with nested cycles, which was rightfully analyzed using our existing feed-forward analysis.

Note that the order in which we considered the three cycles in the preceeding paragraph served only to explain our line of thought. During performance analysis we need to check the validity of the conditions in lemma 6.1 for all cycles, but we can do it in any order. If the conditions are violated for only one cycle, analyzing the corresponding feed-forward system is not a valid analysis approach.

6.11 Cycles with Multiple Inputs

Let us now consider cycles with more than one external activating inputs. This adds a level of complexity since in general the phases between tokens arriving at the two external inputs are not restricted.

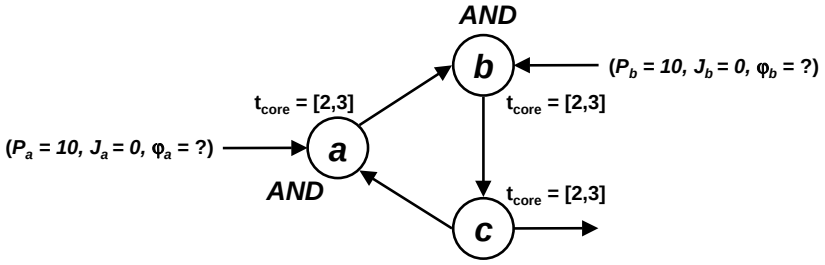


Figure 6.7. Cycle with two external inputs.

Consider the example of a cycle with two inputs in Fig. 6.7. Let us assume that each task has a core execution time interval of $[2, 3]$ and is mapped on an exclusive resource (i. e. response time intervals equal core execution time interval), that the cycle contains one initial token, and that the external event models are ($\mathcal{P}_{a,ext} = \mathcal{P}_{b,ext} = 10$, $\mathcal{J}_{a,ext} = \mathcal{J}_{b,ext} = 0$). What we do *not* know are the phases of events in one external event stream relative to events in the other external event stream.

Even though the sum of worst-case response times around the cycle is 9 and thus less than the external period (10), it is possible that a token arriving at the cycle-external input of task a has to wait for an internal token because of the AND-activation delay of task b, and vice versa. This is illustrated in the two gantt charts in Figs. 6.8 and 6.9.

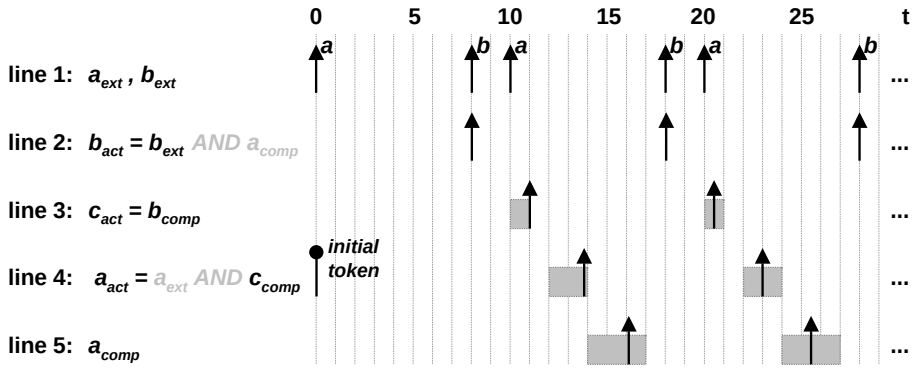


Figure 6.8. Possible event timing and events for our multi-input example (Fig. 6.7) assuming a phase of +8 for cycle-external events at task b relative to task a, and an initial token at the cycle-internal input of a.

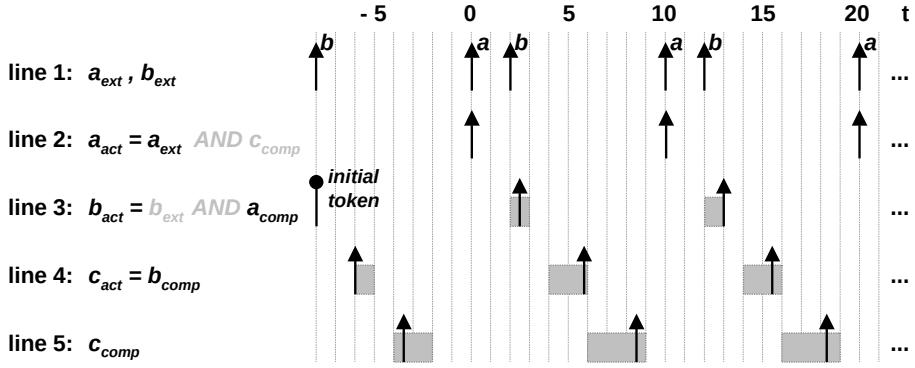


Figure 6.9. Possible event timing and events for our multi-input example (Fig. 6.7) assuming a phase of −8 for cycle-external events at task b relative to task a, and an initial token at the cycle-internal input of b.

In Fig. 6.8 an external token arrives at task b 8 time units *after* the corresponding external token has arrived at task a. We assume that the

cycle token is initially at the cycle-internal input of task a [4]. As can be seen, the schedule quickly settles into a pattern where the activating jitter of task a is $\mathcal{J}_{a,act} = 2$, and is thus larger than the external jitter. On the other hand, the activating jitter of task b is $\mathcal{J}_{b,act} = 0$, and thus equals the external jitter. We observe the opposite effect in Fig. 6.9, where an external token arrives at task b 8 time units *before* the corresponding external token has arrived at task a. We assume that the cycle token is initially at the cycle-internal input of task b [5]. It is also obvious that the additional AND-delay does not lead to unbounded backlog at the cycle-external inputs.

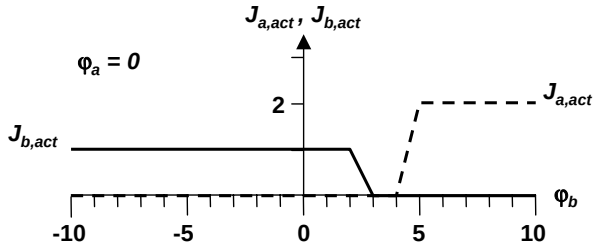


Figure 6.10. Activating jitter of tasks a and b as a function of the phase between external tokens arriving at the two tasks. φ_a is zero, φ_b is varied in the interval $[-10, 10]$.

Fig. 6.10 shows the activating jitter of tasks a and b as a function of the phase between external tokens arriving at the two tasks. φ_a is defined as zero, φ_b is varied in the interval $[-10, 10]$. We observe that at least one of the activation jitters is zero for all phases. We could therefore analyze two corresponding feed-forward systems separately, one where the cycle is cut at the input of task a, the other where the cycle is cut at the input of task b. For each task in the cycle we would keep the smaller minimum response time, and the larger maximum response time.

The generality of this idea has not been investigated further, since the practical relevance of cycles with more than one external activating inputs is doubtful. Should such a system occur in practice, then the designer can be made aware of the complex scheduling dependencies that he is building into his system and guided towards a more robust solution. One such solution would be to replace all but one input by communication registers, which can be read at any time and do not

[4] If the cycle token is initially at the cycle-internal input of task b, then we obtain the same schedule for $t \geq 8$.

[5] If the cycle token is initially at the cycle-internal input of task a, then we obtain the same schedule for $t \geq 0$.

contribute to task activation. An alternative would be to use a single AND-activated task which activates the cycles once all require input data for one cycle iteration is available.

6.12 Multi-Rate Cycles

Due to the data-rate transitions in a multi-rate cycle, token arrival at cycle-internal inputs becomes more irregular than in a single-rate cycle (chapter 5). Therefore, conditions under which analysis of a corresponding feed-forward system is valid may be very conservative. A less conservative solution requires considering possible event timing for at least a macro period (chapter 5). A general solution is beyond the scope of this thesis. However, the idea will be considered for the special case of cycles stemming from Simulink multi-rate designs in chapter 9.

6.13 Summary and Conclusion

In this chapter we considered compositional performance analysis for single-rate cyclic task dependencies, which are typical in control systems and complex filters. We showed that a straight-forward application of our compositional performance analysis approach does not yield a fixpoint. However, by considering phases between events and initial tokens in a cycle we then showed that a significant number of cycles in fact has a fix point such that analysis of an equivalent feed-forward system produces correct results. Furthermore we showed that such cycles can be arbitrarily nested. These are significant results since they allow us to analyze many real-world systems with cyclic task dependencies with only minor extensions to our compositional performance analysis approach (detection of cycles, distinguishing AND-concatenated tasks inside cycles from those outside cycle, performing simple checks).

We provided suggestions how to transform those systems that initially do not satisfy all conditions that allow us to analyze an equivalent feed-forward system. The designer needs to be alerted of the problem, and a solution must be indicated. The potential problems that we identified can be resolved either by increasing the number of initial tokens in the cycle, or by reducing the number of external activating inputs that influence cycle tasks. The decision to modify the system then becomes a trade-off between (perceived) performance and analyzability.

Chapter 7

SYSTEM-LEVEL ANALYSIS EXAMPLE

In this chapter, we apply the results from the previous chapters to analyze the performance of our introductory system. For convenience, the system is shown here again.

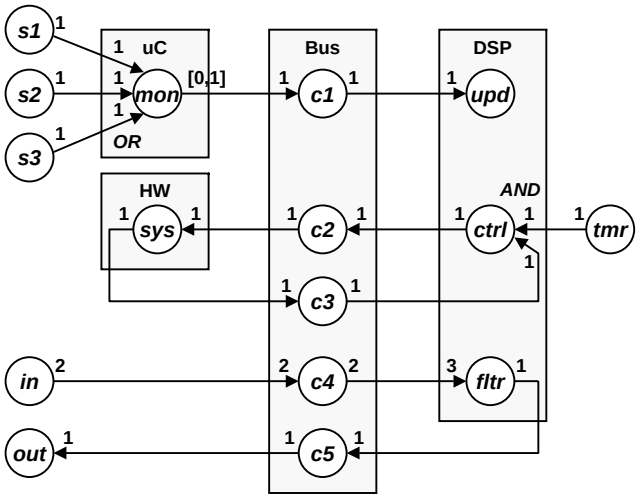


Figure 7.1. Example of an embedded real-time system with a variety of task dependencies

The embedded system in Fig. 7.1 represents a hypothetical SoC consisting of a micro-controller (uC), a digital signal processor (DSP) and dedicated hardware (HW), all connected via an on-chip bus (Bus). DSP and uC are equipped with local memory. The HW acts as an interface to a physical system. It runs one task (sys) which issues actuator com-

mands to the physical system and collects routine sensor readings. *sys* is controlled by task *ctrl*, which evaluates the sensor data and calculates the necessary actuator commands. *ctrl* is activated by a periodic timer (*tmr*) and by the arrival of new sensor data (AND-activation in a cycle). We assume 2 initial tokens in the cycle.

The physical system is additionally monitored by 3 sensors (*s1* - *s3*), which produce data sporadically as a reaction to irregular system events. This data is registered by an OR-activated monitor task (*mon*) on the *uC*, which decides if the control algorithm needs to be updated. If an update is necessary (conditional communication), then this information is sent to task *upd* on the *DSP*, which updated parameters into shared memory.

The *DSP* additionally executes a signal-processing task (*fltr*), which down-samples (data rate transition), filters and compresses a stream of data arriving at input *in*, and sends the processed data via output *out*. All communication, except for shared-memory on the *DSP*, is carried out by communication tasks *c1* - *c5* over the on-chip *Bus*. Core execution times for each task are shown in Tab. 7.1.

computation task	C	communication task	C
mon	$[10, 12]$	*c1*	$[4, 4]$
sys	$[15, 15]$	*c2*	$[4, 4]$
upd	$[5, 5]$	*c3*	$[4, 4]$
ctrl	$[20, 23]$	*c4*	$[8, 8]$
fltr	$[12, 15]$	*c5*	$[4, 4]$

Table 7.1. Core execution and communication times

We assume the following event models at system inputs (Tab. 7.2).

input	s/p	$\mathcal{P}_{in}$	$\mathcal{J}_{in}$	$d_{min,in}$
s1	s	1000	0	0
s2	s	750	0	0
s3	s	600	0	0
in	p	60	0	0
tmr	p	70	0	0

Table 7.2. Event models at external system inputs.

In order to function correctly, the system has to satisfy a set of path latency constraints (Tab. 7.3). Constraints 1 and 3 have been explicitly specified by the designer. The $2nd$ constraint implicitly follows from the

fact that the cycle contains 2 initial tokens (chapter 6). Constraint 3 is defined for causally dependent tokens [123]. We shall also impose a maximum jitter constraint at output *out* (Tab. 7.4).

constraint #	path	maximum latency
1	*s1*, *s2*, *s3* $\rightarrow$ *upd*	70
2	cycle (*ctrl* $\rightarrow$ *ctrl*)	140
3	*in* $\rightarrow$ *out*	120

Table 7.3. Path latency constraints

constraint #	output	event model period	event model jitter
4	*out*	$\mathcal{P}_{out} = 90$	$\mathcal{J}_{out,max} = 60$

Table 7.4. Output jitter constraint (the output period automatically follows from the data rate transition)

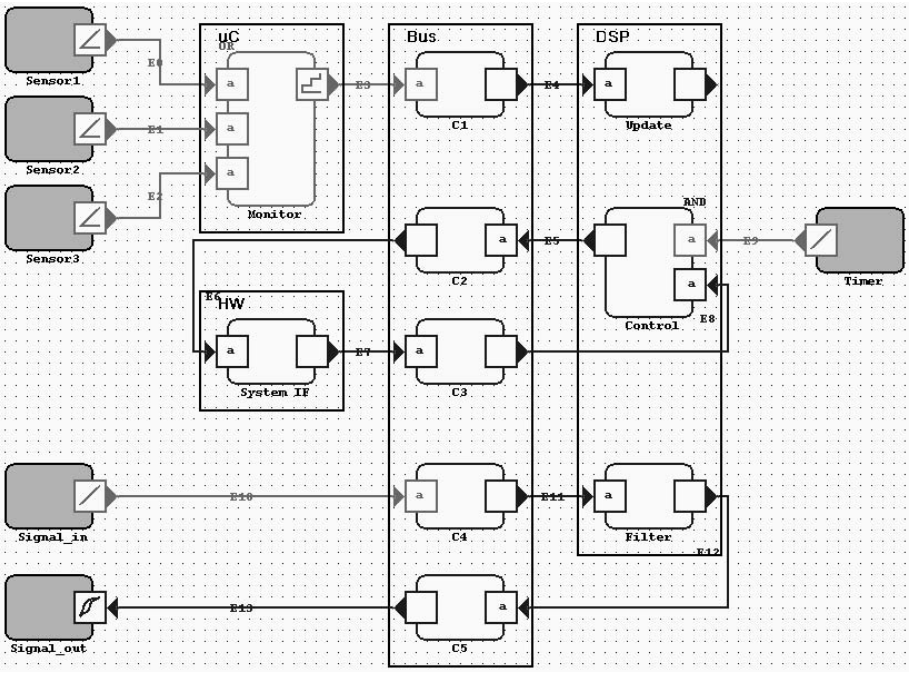


Figure 7.2. Single-rate version of our example system modeled in SymTA/S

Performance analysis results were obtained to a large extent using our system-level performance analysis framework SymTA/S [107]. However, at the time of writing, our data rate transition algorithms had not been integrated into SymTA/S. Therefore, we used a similar single rate systems, and performed the missing calculations using an external program. A SymTA/S screen-shot of the single-rate system is shown in Fig. 7.2.

In the first step, SymTA/S performs OR-concatenation of the output event models of *s1* - *s3* (section 4.2) and obtains the following *sporadic* activating event model for task *mon*: $\mathcal{P}_{act} = \mathcal{P}_{OR} = 250$, $\mathcal{J}_{act} = \mathcal{J}_{OR} = 500$. The large jitter is due to the fact that input events happening at the same time lead to a burst of up to 3 activations (we assume no correlations between *s1* - *s3*). Since task *mon* is the only task mapped onto *uC*, we can now perform local scheduling analysis for this resource, in order to calculate the minimum and maximum response times, as well as the output event model of task *mon*. The results of this analysis are shown in Tab. 7.5. This is also the analysis status displayed in Fig. 7.2.

task	s/p	$\mathcal{P}_{act}$	$\mathcal{J}_{act}$	$d_{min,act}$	C	r	s/p	$\mathcal{P}_{out}$	$\mathcal{J}_{out}$	$d_{min,out}$
mon	s	250	500	0	$[10, 12]$	$[10, 36]$	s	250	502	10

Table 7.5. Scheduling analysis input and output on resource *uC*

The worst-case response time of task *mon* increases compared to its worst-case core execution time, since later activations in a burst have to wait for the completion of the previous activations. The output jitter increases by the difference between maximum and minimum core execution times compared to the activation jitter. The minimum distance between output events equals the minimum core execution time (section 3.6.2).

At this point, the rest of the system cannot be analyzed, because on every resource activating event models for at least one task are missing. SymTA/S therefore generates a conservative starting-point by propagating all output event models along all paths until an initial activating event model is available for each task (section 3.5). The initial activating event model for task *fltr* and all subsequent tasks in the path is obtained by performing rate-transition calculation (chapter 5) on the output event model of source-task *in*. SymTA/S then checks that the system cannot be overloaded in the long term. This calculation requires only activation periods and worst-case core execution times and thus can be done before response-time calculation.

System-level analysis can now be performed by iterating local scheduling analysis and event model propagation (section 3.6.5). SymTA/S determines that task *ctrl* belongs to a cycle, checks that AND-concatenation is selected, and then proceeds to analyze the corresponding feed-forward system. SymTA/S executes until a fix-point for the whole system has been reached, and then compares the calculated performance values against performance constraints[1].

Let us explore different scheduling options with regards to the timing constraints that we specified above. We use static priority scheduling both on the *DSP* and on the *Bus* and vary task priorities. In Tab. 7.6 we show performance analysis result for four experiments with different priority assignments. In the 2*nd* and 3*rd* columns, tasks are ordered by priority, highest priority on the left. In the last four columns, we give the actual values for all four constrained values from Tabs. 7.3 and 7.4, and indicate which constraints are met ($\surd$). A latency along path 3 is only calculated if constraint 4 is met.

exp.	*Bus* tasks	*DSP* tasks	constr. 1	constr. 2	constr. 3	constr. 4
1	*c1*, *c2*, *c3*, *c4*, *c5*	*upd*, *ctrl*, *fltr*	45 $\surd$	77 $\surd$	n/a	119
2	*c1*, *c2*, *c3*, *c4*, *c5*	*fltr*, *upd*, *ctrl*	60 $\surd$	107 $\surd$	n/a	97
3	*c4*, *c5*, *c1*, *c2*, *c3*	*fltr*, *upd*, *ctrl*	74	130 $\surd$	95 $\surd$	41 $\surd$
4	*c1*, *c4*, *c5*, *c2*, *c3*	*fltr*, *upd*, *ctrl*	60 $\surd$	158	111 $\surd$	57 $\surd$

Table 7.6. *Bus* and *DSP* tasks ordered by priority (highest on left), and calculated actual values for all four constrained values from Tabs. 7.3 and 7.4

In the first experiment, only constraints 1 and 2 are satisfied. If we increase priorities of tasks belonging to path 3 until constraints 3 and 4 are met, we violate either constraint 1 or constraint 2 (experiments 3 and 4). At this point, one solution is to increase the speed of one or several components. Let us repeat experiment 3, but this time let us increase the clock-rate of *uC* to 120% of its original clock rate. This will speed up task *mon* which has the largest core execution time of all tasks belonging to path 1 (which violates its constraint in experiment 3). As can be seen in Tab. 7.7, all constraints are now satisfied.

There are several other solutions to satisfy all timing constraints. Let us consider another, particularly interesting one. We will repeat experiment 4 in our set of original experiments, but this time we will *reduce*

[1] In the future, it will also be possible to specify abort conditions, since it cannot be guaranteed that a fix-point can be reached. Currently, a designer has to hit the 'stop' button to abort.

exp.	*Bus* tasks	*DSP* tasks	constr. 1	constr. 2	constr. 3	constr. 4
$3'$	*c4*, *c5*, *c1*, *c2*, *c3*	*fltr*, *upd*, *ctrl*	68 √	130 √	95 √	41 √

Table 7.7. Experiment 3 from Tab. 7.6 repeated with the clock-rate of uC increased to 120% of its original clock rate.

the clock-rate of uC to 80% of its original clock rate. Surprisingly, again all constraints are met (Tab. 7.8).

exp.	*Bus* tasks	*DSP* tasks	constr. 1	constr. 2	constr. 3	constr. 4
$4'$	*c1*, *c4*, *c5*, *c2*, *c3*	*fltr*, *upd*, *ctrl*	69 √	124 √	107 √	53 √

Table 7.8. Experiment 4 from Tab. 7.6 repeated with the clock-rate of uC reduced to 80% of its original clock rate.

The satisfaction of constraint 1 is easily comprehensible. This constraint was already satisfied in experiment 4, and the now slower uC did not increase the latency on path 1 beyond the given deadline. But why did the worst-case latencies on paths 2 and 3, as well as the jitter at output *out* decrease compared to experiment 4? Due to the slower uC, task *mon* produces data with a larger minimum distance $d'_{min,out} = 12.5$. This leads to larger minimum gaps between activations of task *c1*, and consequently to a smaller worst-case transient load by task *c1* on the *Bus*. Since task *c1* interrupts *c2* - *c4*, the larger gaps between these interrupts allow the lower priority task to execute earlier, thus reducing the worst-case latencies on paths 2 and 3, as well as the jitter at output *out*. Alternatively, we could have inserted a d_{min}-EAF (section 3.6.6) between tasks *mon* and *c1* to separate consecutive activating events for *c1*. This might be a better solution if other tasks were sharing uC that would violate their constraints if uC was slowed down.

Reducing transient overload on the bus is an example for a *scheduling anomaly* [122, 97]. Worst-case response time analysis alone would not have caught this situation, since the minimum distance between activations of task *c1* is determined by the *best-case* response time of task *mon*. Performance simulation might not have caught this situation either, since best-cases are often not tested.

In summary, our analysis extensions allow us to reap the full range of benefits of compositional performance analysis in the presence of a variety of task dependencies typical in complex embedded applications.

This is an important step towards applicability of performance analysis in the real world. As was shown, we were able to calculate conservative performance bounds for our example featuring both OR- and AND-concatenated tasks, a functional cycle, conditional communication and data rate transitions. We explored several different scheduling options, identified a scheduling anomaly which would likely have been overlooked during simulation, and found a non-intuitive solution that meets all timing constraints on the target architecture while allowing to reduce the speed of one component.

Chapter 8

CONTEXT-AWARE ANALYSIS

Most of the scheduling analysis techniques considered so far can be rather pessimistic, because they ignore certain correlations between consecutive task activations. For example, worst-case response time analysis typically assumes that every activation of a task leads to the worst-case core execution time of that task. However, in a sequence of activations of the same task, some activations may lead to shorter core execution times. The designer may be able to derive rules from his knowledge of the application which narrow down the execution times in a sequence of task activations. We call such rules *intra event stream contexts*.

Also, worst-case response time analysis typically assumes worst-case transient load distributions. This usually implies the assumption that all tasks sharing a resource can be activated at the same time. However, in reality it may not be possible to activate all these tasks at the same time. Based on his knowledge of the system environment, the designer may be able to narrow down possible phases between activations of different tasks. We call such phase information *inter event stream contexts*.

The idea to improve scheduling analysis by considering some kind of system context is not new. Mok and Chen introduced the idea of *intra* event stream contexts and showed promising results for response-time analysis for the transmission of MPEG-streams, where the worst-case load for I-, P- and B-frames differs considerably [79, 7]. Recently, Thiele et al. extended their compositional performance analysis with load curves which allow them to consider intra event stream contexts [77]. Tindell introduced the concept of *inter* event stream contexts for tasks scheduled by a static priority preemptive scheduler [113]. This work was later generalized by Palencia and Harbour [85]. Inter event stream contexts

are also used in the holistic scheduling analysis by Eles et al. for mixed time-triggered/event-triggered systems [90–92].

Context-aware analysis is another important step towards acceptance of performance analysis for complex embedded applications. Our contribution lies in the generalization of intra event stream contexts, as well as in the combination of both types of contexts during analysis.

8.1 Intra Event Stream Contexts

Context-blind analysis assumes that in the worst case respectively best case, every task executes each activation with its worst-case execution time (WCET) respectively best-case execution time (BCET). In reality, different input data often activates different behaviors of a computation task with significantly different WCETs [121, 120], or can lead to significantly different bus loads for a communication task [126]. Such different behaviors are sometimes called modes [126]. Modes can also be generated internally in a task by a finite state machine which steps into a different state each activation.

If it is possible to provide some rules for the sequence of activations of different modes of a task C (i. e. specify *intra* event stream contexts), then compared to the context-blind case a lower maximum load and a higher minimum load can be determined during scheduling analysis for a sequence of successive activations of the task. This in turn can lead to shorter calculated worst-case response times and longer best case response times of tasks that can be preempted by task C, thus obtaining tighter performance bounds for theses tasks.

In their work, Mok and Chen assume that a periodic sequence of types of activating events is completely known [7, 79]. While this is sufficient to model e. g. a single MPEG-stream with a fixed sequence of I-, P- and B-frames, in general intra event stream contexts can be more complex. For example, an MPEG stream may be encoded in one of several patterns of I-, P- and B-frames, or several MPEG streams may be interleaved. A different example is an OR-activated task (section 4.2) with different behaviors depending on the input at which data is received. In all theses cases, the exact order of different event types is unknown.

Mok and Chen also do not clearly distinguish between different types of events on one hand, and different task behaviors on the other. However, this distinction is crucial for subsystem integration and compositional performance analysis. Different types of events are a property of the sender, while different behaviors are a property of the receiver. Both can be specified separately from each other and later correlated. Furthermore, it may be possible to propagate intra event stream contexts

along a chain of tasks. It is then possible to also correlate the modes of consecutive tasks.

In our extension to Mok's and Chen's work, we allow minimum- and maximum-conditions for the occurrence of a certain type of event in a sequence of a certain length n, in order to capture partial information about an event stream. For example, "min 2 *blue* events out of 6" indicates that in any sequence of 6 events in an event stream, at least two will have a property that we call 'blue' (in allusion to the terminology used for colored petri nets [58]).

Once intra event stream contexts and modes are combined for a consumer task C, a single worst-case and a single best-case sequence of events with length n can be determined from the available min- and max-conditions that can be used to calculate the worst- and best-case load due to any number of consecutive activations of C. We have extended static-priority preemptive response-time calculation to exploit this idea [50]. The following algorithm performs worst-case calculation. A detailed discussion can be found [40].

```
/* generate worst-case token sequence of length n */
1  FulfillAllMinConditions();
2  CompleteSequenceConsideringMaxConditions();
3  SortSequenceByWeigth();

/* peform scheduling analysis */
4  CalculateWorstCaseResponseTimes(); // using equation 8.1
```

Figure 8.1. Algorithm for worst-case response time calculation considering intra-contexts with minimum- and maximum-conditions for the occurrence of a certain type of event.

Our algorithm (Fig. 8.1) first fulfills min-conditions, which specify the smallest number of events of a certain color in a sequence of n events (`line1`). After this step, in order to obtain a worst-case sequence, as many of the remaining places in the sequence as max-conditions allow are first filled with event colors that activate the mode with the largest WCET, then with event colors that activate the mode with the second largest WCET, and so on (`line2`). Then the resulting sequence is sorted by weight, starting with the color that activates the mode with the largest WCET (`line3`).

Intra event stream contexts can now be exploited for scheduling analysis (`line4`). In case of a static priority preemptive scheduler, the following extension of equation 3.5 gives the worst-case response time for

lower-priority task i. $C_{j,k}$ is the WCET of the kth activation of higher-priority task j in a worst-case sequence of $\eta_j^u(r_i)$ activations due to an intra event stream context. If j is a task without intra event stream context information, $C_{j,k}$ equals C_j.

$$r_i = C_i + \sum_{\forall j \in hp(i)} \sum_{k=1}^{\eta_j^u(r_i)} C_{j,k} \tag{8.1}$$

In [50], we compared worst-case response time analysis improvements using intra event stream context information to the context-blind case for a variety of system-bus speeds of a hypothetical set-top-box. An important observation was that the greatest reductions in the calculated worst-case response time (up to 90 %) due to intra event stream contexts are obtained for slow bus speeds. The reason is that for slower bus speeds low-priority traffic is interrupted more often, and thus calculation of larger gaps between high-priority traffic using intra-contexts has a larger influence. This observation is important since the designer usually strives to reduce resource speed as much as possible to save cost and power consumption.

8.2 Inter Event Stream Contexts

Context-blind analysis usually assumes that in the worst-case all scheduled tasks are activated simultaneously. In reality, activating events are often time-correlated, which rules out simultaneous activation of all tasks. For priority-based scheduling, this may lead to a lower maximum number, and a higher minimum number of interrupts of a lower-priority task through higher-priority tasks compared to the context-blind case, again resulting in a shorter worst-case response time, and longer best-case response time of the lower priority task.

Inter event stream contexts capture information about time-correlated events in different event streams in a way that can be exploited by performance analysis. Tindell introduced this idea for tasks scheduled by a static priority preemptive scheduler [113]. Each set of time-correlated tasks is grouped into a so called *transaction*. Each task is activated when an offset elapses after the activation of the transaction to which it belongs. Transactions are activated by periodic external events.

Tindell showed that the worst-case contribution of a transaction to the response time of a lower-priority task occurs when the activation time of the lower-priority task happens as soon as possible after the *critical instant* of the transaction. The critical instant is the activation time of one of the transaction's higher-priority tasks. Subsequent activations of

higher-priority tasks belonging to the transaction also have to happen as soon as possible after the critical instant.

Since all activation times of all higher-priority tasks belonging to a transaction are candidates for the critical instant of the transaction, the worst-case response time of a lower-priority task has to be calculated for all possible combinations of all critical instants of all transactions that contain higher priority tasks, to find the absolute worst-case. I. e. the algorithm is exponential with the number of transactions.

In [50], we compared worst-case response time analysis improvements using inter event stream context information to the context-blind case for a variety of offset values. An important observation was that inter event stream context analysis reveals the non-linear influence that a small local change, in our example the speed of one execution unit, can have on system-performance.

8.3 Combination of Contexts

Inter and *intra* event stream contexts are orthogonal: the worst-case response time of a lower-priority task is reduced both because fewer high-priority task activations can interrupt its execution during a certain time interval, and because the time required to process a sequence of activations of each higher-priority task is reduced. Therefore, performance analysis can be further improved if it is possible to consider both types of contexts in combination.

8.4 Example

Let us apply context-aware scheduling analysis to the system in Fig. 8.2, where three tasks have been mapped onto a static priority preemptive scheduled resource. Task properties are given in table 8.1. The core execution time of task b is an interval. A detailed model reveals that task b has three modes with different core execution times that are activated by events with different colors.

task	$\mathcal{P}_{act}$	C	*priority*
a	100	30	*hi*
b	100	$[15, 25]$ (red = 25, green = 20, blue = 15)	*mid*
c	300	100	*lo*

Table 8.1. Task properties used in our context-aware scheduling analysis example. Task b has three modes with different core execution times that are activated by events with different colors.

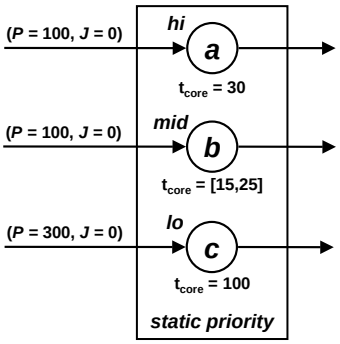


Figure 8.2. Example system for context-aware scheduling analysis

color	min # of events	max # of events
red		2
green	2	
blue	1	1

Table 8.2. Intra event stream context: Min- and max-conditions for event colors in a sequence of 4 events generated by task s_b

Let us suppose that we know min- and max-conditions for a sequence of 4 events generated by source s_b which activates task b (table 8.2). The resulting partial sequence after fulfilling all min-conditions is: ('green', 'green', 'blue'). A color now has to be assigned to the $4th$ event, and the sequence must be sorted. In the worst-case, the $4th$ event is 'red', because it activates the mode of task b with the longest core execution time (table 8.1). The sorted worst-case event sequence of length 4 is: ('red', 'green', 'green', 'blue'). In the best-case, the $4th$ event is 'green'. This event only activates the mode with the $2nd$ shortest core execution time (table 8.1). However, we cannot add another 'blue' event (which activates the mode with the shortest core execution time), because it would violate the max-condition in table 8.2. The sorted best-case event sequence of length 4 is: ('blue', 'green', 'green', 'green').

Let us further assume that we know the phase between activations of tasks a and b: activations of task b occur 50 time units after activations of task a. We can now compare scheduling analysis results for four cases: context-blind, only intra contexts, only inter context, and combination of both contexts. Calculated response times are given in table 8.3.

Neither intra nor inter-contexts can affect task a, since its response time is already a single value in the context-blind case. The response time of task b cannot be affected by its own intra-contexts, since for

task	r_{blind}	r_{intra}	r_{inter}	$r_{intra+inter}$
a	30	30	30	30
b	[15, 55]	[15, 55]	[15, 30]	[15, 30]
c	[145, 265]	[150, 255]	[160, 240]	[165, 235]

Table 8.3. Calculated response times: Context-blind, only intra contexts, only inter context, and combination of both contexts

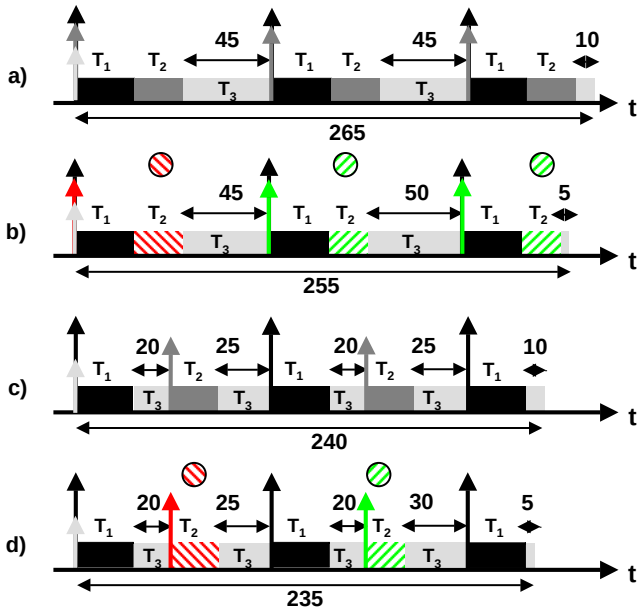


Figure 8.3. Worst-case response time scenarios for task c obtained in the context-blind case (a); analysis improvement due to intra (b), inter (c), and the combination of intra and inter event stream contexts (d)

worst-case (best-case) response time calculation the longest (shortest) core execution time must be considered across all modes, which is equivalent to not considering the intra-context at all. The inter-context reduces worst-case response time, since a phase of 50 guarantees that b cannot be preempted by a. Both the best- and the worst-case response time of task c are affected by both types of contexts, and by their combination. The worst-case response time scenarios for task c for all four cases are shown in Fig. 8.3.

8.5 Summary and Conclusion

In this chapter we introduced intra and inter event stream contexts and demonstrated that considering them can yield significantly tighter performance analysis bounds compared to context-blind analysis. We followed the approach suggested in SPI [126] and separated the modeling of context information from the modeling of task modes. This approach supports the idea of subsystem integration in compositional performance analysis, since contexts and modes can be specified separately and later correlated. We used a simple activation function where the occurrence of an event with a particular color activates a particular mode. This could be extended to more complex activation functions as suggested in SPI, e. g. to consider a combination of colors at different task inputs when determining which mode is activated.

Apart from fully specifying repeating sequences of colors, we additionally consider minimum- and maximum-conditions for the occurrence of a particular color in a sequence of a certain length n, in order to capture partial information about an event stream. This allows us to apply intra-context aware analysis for event streams where the exact order of colors is unknown, e. g. if an event stream is multiplexed from multiple sources. We presented an algorithm which allows us to calculate the worst-case sequence of colors for the activation of a task with known modes. We extended static-priority preemptive response-time calculation to exemplify the applicability of our approach. We borrowed the inter event stream context model and analysis ideas by Tindell and Palencia and showed that they can be nicely combined with intra event stream contexts, since both types of contexts are orthogonal and improve analysis in different ways.

We assumed that intra- and inter-contexts are specified manually by the designer. Future work could derive inter-contexts from the response times of analyzed tasks. We implicitly used this approach to determine under which conditions a cycle-internal token does not influence the activating event model of an AND-concatenated task in a cycle (chapter 6)[1]. Some intra-contexts can also be derived automatically e. g. by analyzing the timing at different inputs of an OR-concatenated task.

[1] Contexts alone would not have done the trick since the concept of initial tokens is also needed

Chapter 9

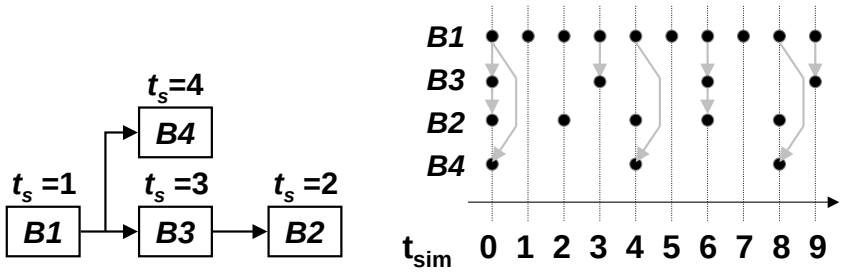
PERFORMANCE ANALYSIS FOR SIMULINK

Simulink [74] is a block diagram oriented industry standard modeling and simulation tool for control and signal processing systems. It builds on the Matlab [73] environment for technical computing. The Simulink extension StateFlow [75] allows to internally model a Simulink block as a state-transition diagram. A typical application of Simulink is the modeling and simulation of a physical system (e. g. from the automotive domain) together with associated control or signal processing functionality. While the model of the physical system is eventually replaced by the actual system, the control or signal processing functionality is implemented as an embedded real-time system.

After an overview of Simulink, we first argue that the scheduling policies that can be synthesized for Simulink designs using existing code-generators are too restricted for state-of-the art applications and architectures. We then show how the Simulink model of computation can be relaxed to enable event-driven scheduling, which is much more flexible, scales well to multi-processor architectures, and is subsystem-integration friendly. Finally, we show that system-level performance analysis is possible for Simulink using our compositional performance analysis approach, albeit with an extended model for phases between different events.

9.1 Simulink Model of Computation

A number of fixed-step and variable-step solvers are available to evaluate a Simulink block-diagram at certain points in time. Variable-step solvers are important in certain situations to model a physical system accurately (e.g. to find zero crossings), but are of little significance for the design of control and signal-processing embedded systems. These appli-



(a) Simulink block-diagram with four blocks with different sample times

(b) Block execution sequence and progress of simulated time

Figure 9.1. Simulink example

cations are generally modeled using *fixed-step solvers* to enable code generation into periodic processes. In the following, only fixed-step solvers are considered.

When a fixed-step solver is used, the Simulink model of computation adheres to the *perfect synchrony hypothesis* [8]. A perfect global clock and strictly periodic block activation are assumed (different periods are allowed for different blocks). While blocks are executed and results are communicated, time does not advance. Time advances only after all computations and communications that need to be performed at a particular clock-tick have been completed. This model is well suited for function simulation, since influences from a real architecture are completely ignored (separation of concerns). Consider the example in Fig. 9.1(a) that shows a simple Simulink block-diagram with four blocks $B1 - B4$, each with a different sample time (or execution period) t_s. The execution timing of the different blocks during simulated time is shown in Fig. 9.1(b). As can be seen, computation and communication are assumed timeless. If multiple blocks need to be executed at the same point in time, then the edges between blocks (Fig. 9.1(a)) define precedence constraints which are obeyed by the simulation engine (indicated by gray arrows in Fig. 9.1(b)).

Simulink blocks additionally can have enabling and/or triggering inputs to allow conditional execution. An enabled block is activated with it sample time, but only executed if the enabling condition is satisfied. A triggered block is activated with the sample time of the block driving the trigger input, but only executed if the triggering condition is satis-

fied. Trigger inputs are of particular interest for blocks that contain a state-transition diagram.

9.2 Code Generation from Simulink

A perfectly synchronous model cannot be implemented on a real architecture. In practice, the implementation must be sufficiently close to simulation. This implies on one hand, that Simulink code generators such as Real-Time Workshop [76] and Target Link [26] have to generate code that allows to compute the same results in the same order as simulation. On the other hand, the target architecture must be able to execute all blocks fast enough with respect to external stimuli (satisfaction of the synchrony hypothesis [8]).

9.2.1 Tick Scheduling

One solution is to execute all computations and communications required at a certain point in time before time has advanced to a value where the next set of computations has to be performed [8]. Real-Time Workshop supports this solution in the so-called *single tasking mode* [76]. In this mode, a tick-scheduler is used to activate each set of computations at the right points in time.

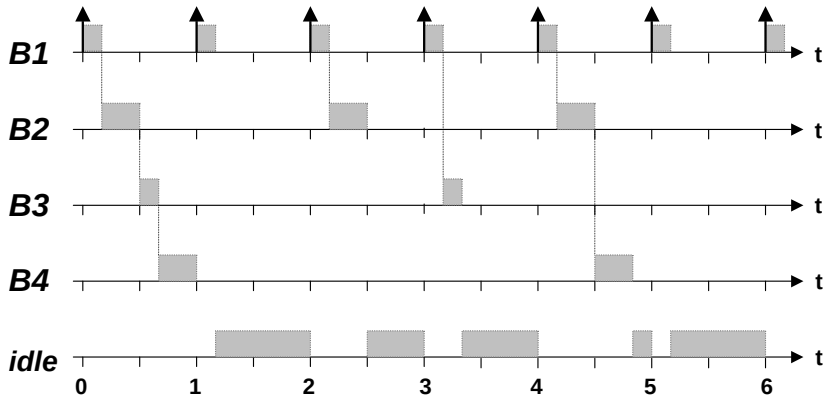


Figure 9.2. Tick scheduling of our Simulink example

A tick-scheduler can be a rather inefficient solution. It must be ensured that the longest sequence of computations that is required at any clock-tick completes before the next clock-tick. This can lead to excessive idle times after short sequence of computations. In our example, it must be possible to execute all four blocks during one clock-tick, as this is required for $t = 0$, $t = 12$ and so on. During all other clock ticks, the

load is considerably smaller (*idle* line in Fig. 9.2). For the core execution times assumed in our example, processor utilization is only 47.22%.

9.2.2 Rate Monotonic Scheduling

An improved solution is to assign priorities to tasks according to their execution rate. This is generally called rate monotonic scheduling (RMS, section 3.2). Real-Time Workshop supports this solution in the so-called *multi-tasking mode* [76].

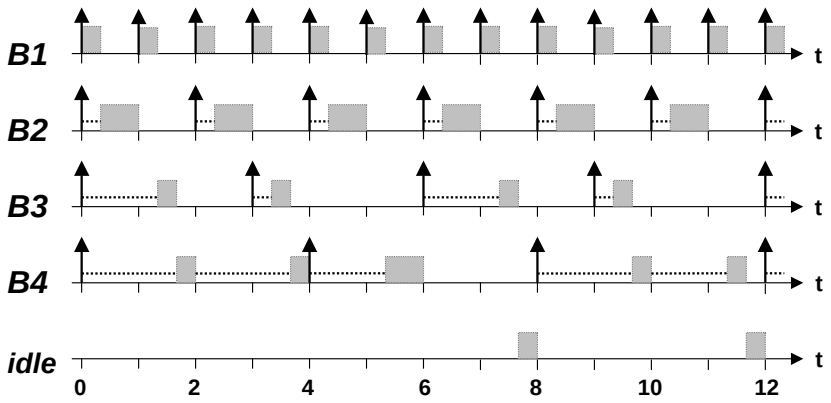


Figure 9.3. RMS scheduling of our Simulink example

In RMS, it must only be ensured that a task with a particular execution rate has been completed before the next activation of the task. Compared to tick-scheduling, load peaks can be processed over a longer stretch of time, thus balancing the load that has to be processed during any time interval. This is shown for our set of blocks in Fig. 9.3. Note that the processor clock-speed has been halved compared to tick-scheduling (different time scales in Figs. 9.2 and 9.3), and that the system is still schedulable. For the core execution times assumed, now only $B1$ and $B2$ can be fully executed during one clock tick. The execution of $B3$ and $B4$ is spread over several clock ticks. However, all tasks meet their deadline. Resource utilization has increased to 94.44%. In reality, the performance improvement due to RMS is diminished somewhat by the RTOS-overhead (section 2.4.1).

However, RMS introduces a new problem [76]. If a faster task provides data to a slower task, then the faster task may produce new output values before the slower task has finished reading all previous values. Data integrity is thus not guaranteed. Likewise, if a slower task provides data to a faster task, then the faster task may start executing before the slower task has provided all necessary values. Both effect can be seen in

Fig. 9.3, e. g. at $t = 1$ between $B1$ and $B3$ (fast $\rightarrow$ slow), and at $t = 0$ between $B3$ and $B2$ (slow $\rightarrow$ fast).

To solve these problems, Simulink provides *rate transition* blocks [76]. These blocks buffer values between faster and slower blocks in a way that data consistency is guaranteed. Rate transition blocks require additional memory for buffering data, and in case of a transition from a slower to a faster execution rate, delay the propagation of data and thus may change the dynamics of a system.

9.3 Simulink Code Generation Issues

Even though preemptive scheduling for Simulink improves on tick-scheduling, both approaches have several limitations which render them poorly suited for complex architectures and applications.

9.3.1 Multi-Processor Implementation

As explained in section 2.1.2, a major problem with global synchrony is that it does not scale well to large architectures. In the case of Simulink, the available schedulers require that each task can be activated periodically, and that activations cannot backlog. This is difficult to achieve for complex multi-processor architectures where arrival times of data can jitter considerably. For example, the automotive industry, which often models in Simulink, is trying to move away from many insular single-processor control systems towards a global view of a networked multiprocessor system that allows them to distribute functions more efficiently [115]. In section 2.4.1 it was explained that an efficient multi-processor implementation of complex applications is only possible if task scheduling is reactive. This is neither the case for tick-scheduling nor for rate-monotonic scheduling.

Another inefficiency stems from the fact that code generators cannot decide for which Simulink blocks strict periodicity is most important. Often, periodicity is only required at sub-system inputs and outputs, while internally the timing is less restricted. Furthermore, even at inputs and outputs it may be acceptable to diverge from a strictly periodic execution up to a certain jitter. Finally, certain paths through a sub-system may be more time-critical than others. There thus exists a variety of timing constraints which cannot be specified in Simulink.

Today, in order to satisfy timing constraints, the designer can manually try to tweak the execution order for a set of blocks within the limits imposed by the existing schedulers. This is tedious and may have to be repeated often, since such restrictive schedules are sensitive to design changes (section 2.1.2). Therefore, it would be much more desirable if

the restrictions imposed by perfect synchrony could be replaced by a set of timing constraints that exactly specify which timing really matters (compositional approach). This novel freedom would provide more scheduling options, thus increasing the potential for an optimized implementation that is also robust to small design changes. Ideally, a chosen schedule could be validated against timing constraints using scheduling analysis techniques.

9.3.2 Multi-Language Design

As explained in section 2.1.4, complex applications are often designed using multiple languages or tools. The different sub-functions then have to be integrated on the target architecture. Function integration can easily violate assumptions made during tick-scheduling or RMS of Simulink designs. For example, preemption can easily corrupt a tick-schedule, since execution of a set of blocks may no longer be complete before the next clock tick. TDMA scheduling, where the tick-schedule is assigned a sufficiently large time-slot may be possible, but this only aggravates the inefficiencies inherent to time-driven scheduling (section 2.4.1), since an even larger worst-case number of tasks must be executable in a short time window.

RMS is better suited for integration with other applications, since it already assumes a priority-based scheduler. However, rate-monotonic priority assignment by no means guarantees timely completion, and a previously valid schedule can easily be invalidated by additional tasks with a sufficiently high priority. We again conclude that in dynamic, reactive systems, more flexible scheduling policies are better suited to resolve such resource conflicts.

9.4 Model Relaxation

It has been established in the previous section that scheduling policies supported by current Simulink code-generators quickly reach their limits when it comes to complex, dynamic, or multi-language systems. Consequently, it would be desirable to preserve the synchrony hypothesis during function design, but relax it in a way during implementation that multi-processor implementation, reactive scheduling, and language integration are supported.

We present an idea where the time-driven activation of Simulink blocks is replaced by data-driven (and hence reactive) task activation. The following aspects of Simulink have to be preserved: 1) relative execution rates between Simulink blocks, and 2) causality and the resulting partial ordering of Simulink blocks, including read and write access se-

quences on each edge to model the register semantics used by Simulink for communication. The SPI model ([126] and section 2.6) will be used to express the transformed Simulink design. SPI allows to efficiently capture the activation dependencies that we are interested in, while abstracting functional details which do not directly influence performance.

9.4.1 Single-Rate Designs

When two Simulink blocks A and B with the same sample rate communicate, then it is possible to replace the communication register by a communication FIFO. This implies that the execution sequence between the two blocks is no longer fixed to $ABAB\ldots$. Instead, A can be executed several times, and the output values buffered until they are consumed by B. This scheduling flexibility comes at the cost of a larger communication buffer and delay. However, buffers and delays can be easily bounded by additional constraints (including a buffer size of one, which then enforces the original, alternating schedule).

9.4.2 Multi-Rate Designs

When two Simulink blocks A and B with different sample rates communicate, then the simple FIFO-solution from the previous section is no longer applicable. In case of a rate transition from a slower to a faster sample-rate, a value may be read more than once. In case of a rate transition from a faster to a slower sample-rate, a value may not be read at all. Both effects cannot be implemented using a FIFO. Therefore, we use SPI register communication to accurately capture the Simulink destructive write, non-destructive read semantics. One token is written (read) on the register channel per activation of the writing (reading) process.

Since a register channel can be read at any time, it cannot contribute to the activation function of a reading SPI process (section 2.6). Therefore, a pair of FIFO-queues is additionally inserted between every two processes with different sample times that communicate over a register channel [48]. Activation of the generated SPI processes is enabled by availability of tokens on those FIFO-queues. Relative execution rates and partial ordering between Simulink blocks are maintained by writing (reading) the appropriate number of tokens to (from) each queue, and by the number of initial tokens on each queue, as specified in the following equations.

$$r(P_i) = t_s(B_i) \tag{9.1}$$

$$n_{C_j(P_{wr}\rightarrow P_{rd})} = r(P_{rd}) - 1 \tag{9.2}$$

$$n_{C_j(P_{rd}\rightarrow P_{wr})} = r(P_{wr}) \tag{9.3}$$

$r(P_i)$ is the number of tokens written and read by process (P_i) per execution on each of its FIFO channels. $t_s(B_i)$ is the sample time of block i. n_{C_j} is the number of initial tokens on FIFO queue C_j. The direction of queue C_j is indicated by indices P_{wr} and P_{rd}, which refer to the writing and reading processes of the corresponding register channel. If sample times are not integer values, they have to be multiplied by an appropriate factor to obtain integer values $r(P_i)$ for each process. Likewise, if sample times have a common divisor, they can be canceled down.

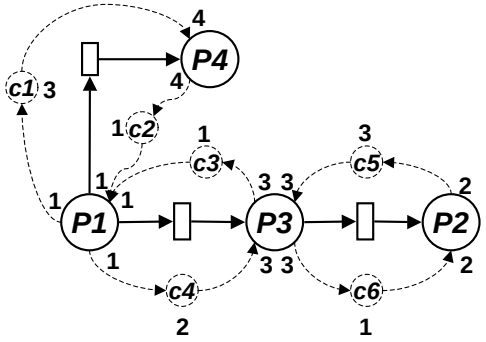
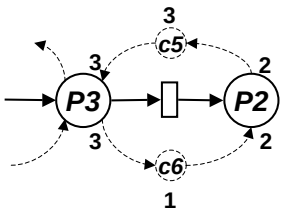


Figure 9.4. SPI representation of the Simulink example from Fig. 9.1(a). Dashed elements are virtual



		tokens in virtual FIFOs after firing	
step	firing	*c5*	*c6*
0		3	1
1	*P3*	0	4
2	*P2*	2	2
3	*P2*	4	0
4	*P3*	1	3
5	*P2*	3	1

Figure 9.5. Valid execution sequence for processes $P2$ and $P3$

Let us apply these rules to our Simulink example in Fig. 9.1(a). The SPI representation of the system is shown in Fig. 9.4. Large circles indicate processes. The number of tokens written or read per execution is annotated to each process port. Small circles indicate FIFO-queues. The number of initial tokens is annotated. Squares indicate register channels. For example, each execution of process $P3$ consumes 3 tokens on channels $c4$ and $c5$, produces 3 tokens on channels $c3$ and $c6$, and

reads and writes the connected registers. Fig. 9.5 focuses on processes $P2$ and $P3$. The table shows the only valid execution sequence for these two processes resulting form their production/consumption rates and from the initial number of tokens on channels $c5$ and $c6$. The sequence repeats periodically after the 5th step and is consistent with the order in which the corresponding Simulink blocks are simulated (Fig.9.1(b)). Possible execution sequences for the complete example in Fig. 9.4, or for any other Simulink design, can be derived accordingly.

9.4.3 Implications for Scheduling

As can be seen, the exact timing of processes is no longer specified in the SPI-representation. Essentially, we have transformed the time-driven Simulink model of computation into a synchronous dataflow representation (section 2.1.1). The only difference to SDF is that the FIFO-queues are not used for communication between processes. They merely represent scheduling constraints, and consequently are modeled as *virtual* SPI elements (section 2.6). This does not hinder the application of a variety of available SDF-scheduling techniques. For example it is now easy to distribute processes to different processors to exploit application parallelism, and to optimize schedules for throughput, latency, memory consumption etc.

If we choose a dynamic scheduling policy, tasks generated from communicating Simulink blocks with different sample times cannot interrupt each other, since at any time, only one of them can be active. This is because the total sum of tokens in a cycle is $r(P_{wr}) + r(P_{rd}) - 1$. However, for simultaneous activation of both processes, $r(P_{wr}) + r(P_{rd})$ tokens would be needed. Since communicating tasks with different sample times cannot interrupt each other, there is no need for rate transition blocks, which were needed for RMS (section 9.2.2).

Reactive scheduling is much more flexible than RMS. Due to activation by availability of data, it does not matter if the timing of input data jitters. A task can only start executing once all tasks that it depends upon have completed. Let us consider the example in Fig. 9.6. Let us assume that $B1$ is externally activated periodically with a certain jitter. A sample sequence of external input events is shown in line *ext* in Fig. 9.6. Let us further assume that the path $B1 \rightarrow B4$ is time critical, and that therefore task priorities have been assigned (highest first): $B1$, $B4$, $B3$, $B2$.

Compared to Fig. 9.3, activation timing of tasks in Fig. 9.6 is more irregular (including bursts of 2 activations for $B2$), both due to the external jitter and the data-dependencies in the system. For example, the activation of $B1$ can occur later than an external event, if the previous

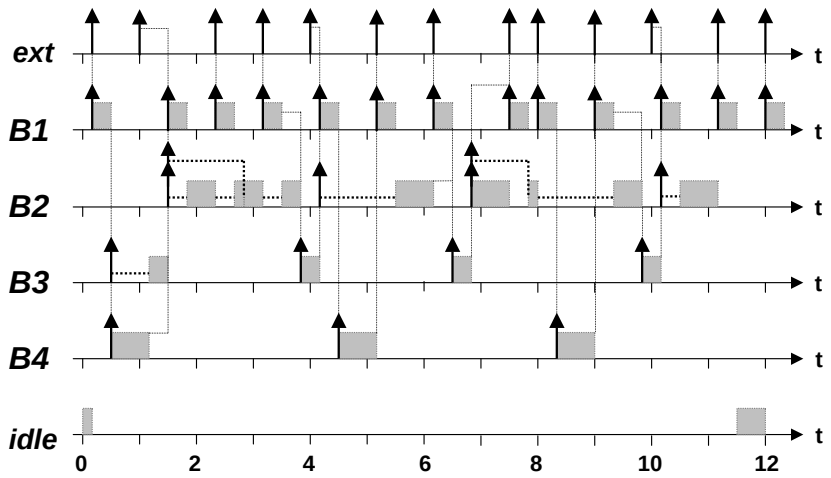
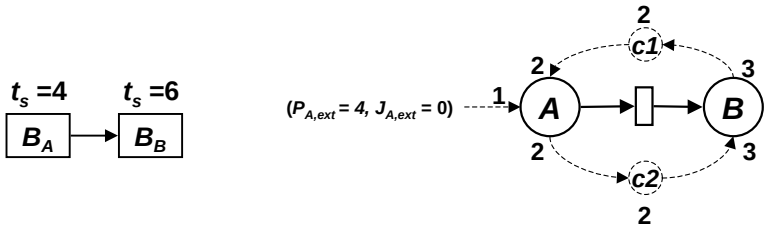


Figure 9.6. Reactive scheduling of our Simulink example

executions of $B3$ and $B4$ have not been completed. No rate transitions blocks are required, since data-dependent tasks do not interrupt each other (but $B1$ and $B4$ can interrupt $B2$). The path $B1 \rightarrow B4$ is executed as quickly as possible, much faster than in the RMS case. The processor utilization is as good as for RMS[1]. We could have chosen a different priority assignment, or even a different scheduling policy depending on the input event timing and timing constraints that need to be satisfied. This flexibility is simply not available with RMS.

One remaining issue is scheduler synthesis. In [48] we showed how to synthesize cyclo-static schedules for multi-rate Simulink designs. In case of dynamic scheduling, the scheduler needs to keep track of the number of produced and consumed tokens on the virtual FIFO channels. Locally, this could be achieved by simple counters. If communicating processes are mapped to different resources, then the number of produced virtual tokens could be made part of a handshake communication protocol. Scheduler synthesis is not considered further in this thesis. It is however essential to exploit the benefits of reactive scheduling and should be developed in close cooperation with a company that offers code-generation for Simulink.

[1]Even better in general, since RMS may become un-schedulable if processor load increases above 69% [68]



(a) Simulink model

(b) SPI representation (sample times have been canceled down to obtain token rates)

Figure 9.7. Example used to explain our analysis approach

9.5 Scheduling Analysis

Like any other scheduling policy, reactive scheduling does not guarantee that all timing constraints are met. Therfore, we would like to analyze the possible timing of a reactive Simulink implementation using the SymTA/S approach. This is straight-forward for single-rate systems, where the tasks generated from Simulink communicate via FIFO buffers (section 9.4.1). However, if two Simulink blocks with different sample times communicate, we obtain a cyclic dependency with two data rate transitions (section 9.4.2). So far, we have only studied data rate transitions without cycles (chapter 5), and cycles without data rate transitions (chapter 6). Both shall now be combined for the special case of tasks generated from Simulink.

In chapter 6 it was shown that even for single-rate cycles a naive approach where only event models are propagated along the cycle without considering further event timing correlations does not yield an analysis fix-point. Therefore, let us follow the approach from chapter 6 and try to find a fix-point for our multi-rate cycle problem by additionally considering the correlations between the arrival timing of different tokens in the cycle.

The Simulink example in Fig. 9.7(a) shall be used for illustration. It consists of two blocks B_A and B_B. B_A has a sample time of 4 time units, B_B has a sample time of 6 time units. The corresponding SPI representation with tasks A and B is shown in Fig. 9.7(b). We assume that an input event at the cycle-external input of task A occurs periodically every 4 time units with a jitter of 1 time unit. We are interested in calculating the activating event models, response time intervals and output event models for both tasks.

9.5.1 Schedulability Condition

Due to the data rate transitions, not every token arrival leads to an activation of tasks A or B. Therefore, let us consider one complete macro period, after which all tokens are in their initial buffers. The following is a sufficient condition to avoid overload: let n_A be the number of executions of task A, and n_B the number of executions of task B during one macro period. In our example, the macro period is $ABABA$, i.e. $n_A = 3$ and $n_B = 2$. An input event at the cycle-external input of task A occurs on average every $\mathcal{P}_{A,ext}$ (4 in our example). Consequently, a new macro period needs to be started on average every $n_A * \mathcal{P}_{A,ext}$ (12 in our example). Since A and B can never be executed at the same time due to data dependencies, the cycle is definitely schedulable with bounded external buffers if the sum of worst case response times of tasks A and B during one macro period does not exceed the average time between starts of the macro period. I. e.

$$n_A * \mathrm{WCRT}(A) + n_B * \mathrm{WCRT}(B) \overset{!}{\leq} n_A * \mathcal{P}_{ext} \tag{9.4}$$

9.5.2 Analysis Approach

As was the case in chapter 6, let us initially assume that the activation timing of task A is solely determined by tokens arriving at the cycle-external input. For our example this implies ($\mathcal{P}_{A,act} = \mathcal{P}_{A,ext} = 4$, $\mathcal{J}_{A,act} = \mathcal{J}_{A,ext} = 1$). Let us additionally define that at system-startup, the first external input event of task A occurs during $t = [0, 1]$, the second during $t = [4, 5]$, and so on (Fig. 9.8, line 1).

Let us assume that scheduling analysis yields a response-time interval of $[1, 2]$ for task A. The resulting output event model is then ($\mathcal{P}_{A,out} = 4$, $\mathcal{J}_{A,out} = 2$). Consequently, task A completes the first execution during $t = [1, 3]$, the next execution during $t = [5, 7]$, and so on (line 2).

Due to the data rate transition between tasks A and B, the activating event model for task B becomes ($\mathcal{P}_{B,act} = 6$, $\mathcal{J}_{B,act} = 4$) (section 5.1). Let us assume that scheduling analysis yields a response-time interval of $[1, 3]$ for task B. We now need to check the schedulability condition in inequation 9.4, which is satisfied: $3 * 2 + 2 * 3 \leq 3 * 4$.

The output event model of task B is ($\mathcal{P}_{B,out} = 6$, $\mathcal{J}_{B,out} = 6$). If we simply calculated the cycle-internal input event model for task A from the output event model of task B using the results from section 5.1, we would obtain an internal input jitter $\mathcal{J}_{A,int} = 10$, invalidating our initial assumption. Such an approach will not yield a fix-point. Therefore, we will consider event timing in more detail.

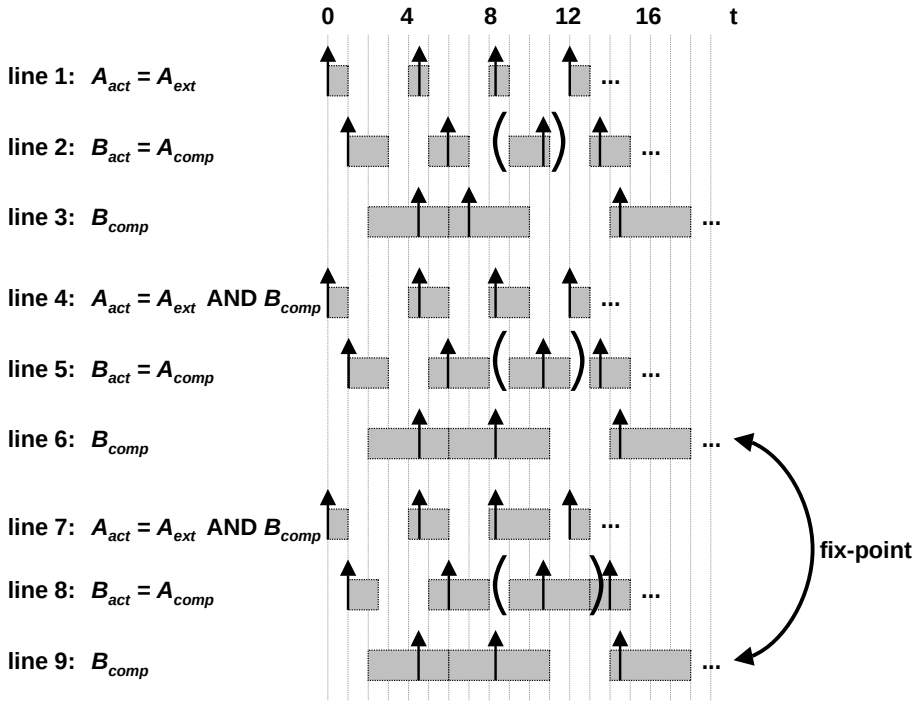


Figure 9.8. Possible event timing as calculated during scheduling analysis of our multi-rate Simulink example

The first and second completions of task A each lead to one activation of task B. Consequently, task B completes its first execution during $t = [2, 6]$, and the second execution during $t = [6, 10]$. The third completion of task A does *not* activate task B (indicated by parentheses in line 2).

The second and third activations of task A depend on the first and second completions of task B, respectively. Therefore, the second and third activation of task A can be delayed by up to one time unit compared to our initial assumption (line 4). This means that our initial assumption was *wrong*. We have to repeat scheduling analysis with a modified activating event model for task A. For this purpose, we need a single activation jitter. To be conservative, we use the largest of all individual activation jitters, yielding a new activating event model ($\mathcal{P}_{A,act} = 4$, $\mathcal{J}_{A,act} = 2$) for task A.

For simplicity, let us assume that the modified activating event model does not change the response time interval of task A. The new possible timing of completions of task A is shown in line 5. A conservative output

event model is ($\mathcal{P}_{A,out} = 4$, $\mathcal{J}_{A,out} = 3$), which results in an activating event model ($\mathcal{P}_{B,act} = 6$, $\mathcal{J}_{B,act} = 5$) for task B. Let us assume that the modified activating event model also does not change the response time interval of task B, yielding the possible completion times shown in line 6.

From line 7 we see that we now have to modify the activating event model for task A to ($\mathcal{P}_{A,act} = 4$, $\mathcal{J}_{A,act} = 3$) to capture the largest jitter of all activations. The resulting completion timing of task A is shown in line 8. Compared to line 5, only the output jitter of the third execution of task A has increased. Since the third execution of task A does not activate task B, the activating event model for task B remains valid. Therefore, a fix-point has been found. The output event model of task A is conservatively assumed as ($\mathcal{P}_{A,out} = 4$, $\mathcal{J}_{A,out} = 4$). This event model is propagated to any other task which might be activated by A.

The final step is to obtain a *periodic with jitter* output event model for task B. $\mathcal{P}_{B,out}$ is obviously 6. Since the maximum distance between two completions of task B is $d_{max} = 18 - 6 = 12$, and the minimum distance between two completions of task B is $d_{min} = 6 - 6 = 0$, we obtain ($\mathcal{P}_{B,out} = 6$, $\mathcal{J}_{B,out} = 6$) (section 3.1.1).

Let us review the extensions to our compositional performance analysis that were required to arrive at these results. Instead of only propagating event models, we additionally propagated the possible phase of each event during one macro period, and exploited our knowledge of task dependencies to arrive at a tight activating event model for task A. Other extensions were not necessary. Iteration to reach a fix-point is already a key existing part of our approach. The calculation of activating event models in the presence of data rate transitions (section 5.1) was reused on the feed forward path from task A to task B. Most importantly, we did *not* need to extend local scheduling analysis techniques, since phase information was not required for this step. Consequently, any available technique can be used, and our compositional approach is not compromised.

Let us now discuss the generality of the presented approach. The approach is independent of the size of the external jitter, and also remains valid if different response time intervals for a task are calculated for different iterations of our algorithm. In the latter case, the schedulability condition (inequation 9.4) has to be checked each iteration. The algorithm will always terminate with one of two possible outcomes. Either a fix-point is found, or the schedulability condition is violated.

The approach has to be extended for situations where a Simulink block A is connected to multiple Simulink blocks with sample-rates different from A (as e. g. in Fig. 9.1(a)). In that case, the task generated

from A has more than two AND-concatenated inputs, which must all be considered when calculating the possible activation timing of A. A much simpler solution is to decouple neighboring sample-rate transitions by inserting dummy tasks such that each task has at most one neighbor generated from a block with a different sample rate. Communication between task and dummy task is via a FIFO-channel in accordance with section 9.4.1. The FIFOs are responsible for decoupling. This solution is an attractive trade-off between buffer-size versus analyzability (design for analyzability). An example is shown in Fig. 9.9 for our initial Simulink system (compare to Fig. 9.4).

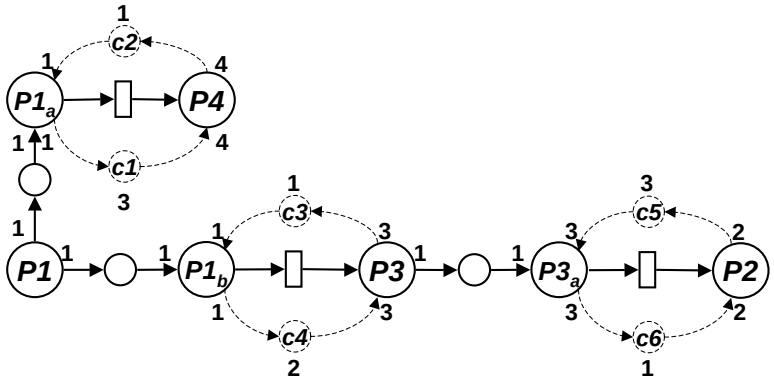


Figure 9.9. SPI representation of the Simulink example from Fig. 9.1(a) with decoupled cyclic task-dependencies.

9.5.3 Triggered and Enabled Blocks

Triggered and enabled blocks are modeled as processes with two modes (section 8.1): In one mode, the triggering/enabling condition evaluates to true and the block is executed. In the other mode, the triggering/enabling condition evaluates to false and nothing else happens. Without further knowledge, during scheduling analysis the core execution time of a triggered/enabled task is assumed to be an interval covering both modes. If intra-context information exists on the occurrence of true/false conditions, then this can be taken into account during scheduling analysis as explained in section 8.1.

FIFO-communication between blocks with the same sample time (section 9.4.1) is not allowed if one of the blocks is a triggered or enabled block, since the FIFO-buffer could over- or underrun. Instead, the solution for multi-rate designs (section 9.4.2) can be used with unit producer/consumer rates on the virtual FIFOs between the blocks. These FIFOs are written/read in both modes and thus are accessed in the same

way as in the case of regular Simulink blocks. The reading/writing of the communication register can be conditional, since it is not buffered and does not contribute to activation.

9.6 Summary and Conclusion

In this chapter we relaxed the perfectly-synchronous Simulink model of computation into an SDF-like representation, which allows reactive scheduling. We then showed the advantages of reactive scheduling compared to the standard tick-scheduling or rate-monotonic scheduling. In particular, it becomes much easier to handle activation with jitter, and to optimize a schedule to satisfy timing constraints. Both benefits are particularly valuable for complex applications and architectures. The relaxation step is applicable for arbitrary Simulink designs.

In the second part of the chapter, we showed that standard scheduling analysis techniques combined with event-model calculation in the presence of AND-activation and data-rate transitions can be re-used to analyze the performance of a reactive system obtained from Simulink. This is an important result since all these analysis techniques were developed without a specific modeling tool in mind. The only, albeit non-trivial extension introduced specifically for Simulink was a more detailed modeling of possible event timing during one macro period of neighboring Simulink blocks with different sample-times. This allowed us to discover correlations and find a fix-point. This extension requires far less work than a hypothetical, non-reusable scheduling analysis framework designed specifically for Simulink.

We provided a suggestion how to transform a system with multiple neighboring sample-rate transitions to enable analysis. As was the case for single-rate cycles, the designer needs to be alerted of the problem, and a solution must be indicated. Again, the decision to modify the system becomes a trade-off between (perceived) performance and analyzability.

Chapter 10

SUMMARY AND OUTLOOK

In this thesis, we presented an analytical approach for performance verification of embedded systems. Our main contribution is a methodology that enables reusing existing analysis techniques to analyze the performance of complex embedded applications.

Embedded applications exhibit a variety of activation dependencies between tasks. Of particular importance are tasks with multiple activating inputs, data-rate transitions between tasks, conditional communication, and cyclic task dependencies. During implementation, applications are mapped onto heterogeneous multi-processor architectures, where a variety of scheduling policies controls access to computation and communication resources shared between tasks. This leads to a large number of non-functional dependencies which affect performance. While existing scheduling analysis techniques capture scheduling dependencies in great detail, they mostly use an over-simplified application model where one activation of a task depends on exactly one execution of exactly one predecessor task. These techniques are thus not applicable for complex embedded applications. In particular, this is a restriction of a novel compositional performance analysis approach developed by our group. This approach is one of the first to enable performance analysis for heterogeneous multi-processor architectures in a scalable way that supports subsystem-integration, making it an attractive choice for large systems.

In order to extend our compositional performance analysis approach for complex embedded applications, we developed transformations in this thesis between the variety of application-level task dependencies, and the possible timing of activating events at the implementation level as required by scheduling analysis. In particular, we showed how to calculate activating event timing in the presence of data rate transitions

(with fixed rates and rate intervals), for tasks with multiple activating inputs (AND- or OR-concatenated), and in the presence of cyclic task dependencies. These transformations allow us to apply available scheduling analysis techniques without modification, and to maintain analysis composability for heterogeneous architectures. In other words, we make available all benefits of compositional performance analysis for complex embedded applications with a small number of additional, re-usable steps. If buffering of communicated data is required due to AND-activation or rate-transitions, then we additionally calculate required buffer sizes as well as minimum and maximum buffering delay.

We derived a set of conditions under which analysis of cyclic task dependencies can be replaced by analysis of a corresponding feed-forward system. This is an enabling step, since it avoids keeping track of possible phases between activations of all cycles tasks over multiple cycle iterations. We provided suggestions how to transform systems that initially do not satisfy these conditions. In a practical analysis flow, the designer needs to be alerted of this situation, and a solution must be indicated. The decision to modify the system then becomes a trade-off between (perceived) performance and analyzability.

We further showed that analysis tightness can be improved considerably by modeling and evaluating different types of system contexts. Specifically, we considered intra and inter event stream contexts, and their combination. Contexts can be specified by the designer based on his application knowledge, and represent another important step towards analyzability of complex applications.

The theoretical results of this thesis have been implemented to a large extent in our compositional performance analysis framework SymTA/S. We used SymTA/S to analyze a SoC featuring both OR- and AND-concatenated tasks, a functional cycle, conditional communication and data rate transitions. We explored several different scheduling options, identified a scheduling anomaly which would likely have been overlooked during simulation, and found a non-intuitive solution that meets all timing constraints on the target architecture while allowing to reduce the speed of one component.

Finally, to demonstrate a link to real-world application modeling, we showed how to apply our performance analysis extensions to designs specified in the industry-standard tool Simulink. We first relaxed the perfectly-synchronous Simulink model of computation into a dataflow representation, which allows reactive scheduling, and is thus well-suited for implementation on multi-processor architectures. We then applied our results for AND-activation and data-rate transitions, extended by a more detailed model of possible activation timing.

Where do we go from here? The conservative nature of performance analysis is often a cause of concern. It is correct that capturing of possible event timing in standard event models may incur a loss of information, both at system inputs and during event model calculation. In particular, construction of *periodic with jitter* or *sporadic with jitter* event models for OR-concatenated tasks or in the presence of data-rate transitions is conservative. One remedy is to use more detailed event models where appropriate. However, it is not sufficient to specify such models at system inputs. It must be ensured that these models can be internally calculated and propagated as well, otherwise they are of little use for analysis composition. More complex models require more complex analyses which need to be developed first, a tradeoff that must be carefully considered. A promising research direction is extensive consideration of contexts beyond the first steps presented in chapter 8. For example, inter-contexts can be used to obtain tighter activating event models for AND- and OR-concatenated tasks. A second suggestion are separate event model parameters for upper and lower event functions, to more accurately model possible event timing in the presence of conditional communication or a combination of periodic and sporadic events (sections 5.2 and 4.2.4). In general, any event model and scheduling analysis extension can be incorporated in SymTA/S, but it should not compromise composability.

Further extensions include the ability to model and analyze additional architecture components with sufficient detail. Important research directions in this area include the analysis of shared memory and cache-related preemption delay, both of which are currently not considered in SymTA/S. On the application side, interesting issues include the consideration of soft constraints, and analysis of transitions between modes of operations, including system startup.

Independent of its current limitations, we believe that in the near future the intractable number of performance dependencies in complex systems will encourage designers to accept performance analysis to reap the benefits of a reliable implementation. Furthermore, we believe that the perceived over-conservative nature of performance analysis is a misconception when viewed from a global perspective. One one hand, local worst-cases do indeed occur at the same time, leading to the calculated global worst case. The poor reliability of modern embedded systems is a sad reminder of this fact. On the other hand, critics of performance analysis often forget how conservatively some system parts are designed today. For example, average load on an SoC bus is rarely more than 30%, in order to have enough headroom for transient load peaks. However, it is often not clear if that is really enough, or if, on the contrary,

precious bandwidth is wasted. A systematic performance analysis approach would provide an upper limit on how bad those peaks really are, thus allowing to reliably dimension bus bandwidth. Furthermore, our compositional performance analysis approach allows to smooth out these peaks, and calculates required buffer sizes and delays incurred in the process (chapter 7).

The tremendous speed of performance analysis, and its applicability early in the design-flow may prove to be its biggest advantages over performance simulation. The ability to perform design-space exploration early allows to optimize an implementation in ways that would not have been considered with simulation for mere lack of time, thus leading to superior solutions which more than compensate the conservative nature of analysis. Therefore, coupling SymTA/S with an exploration framework presents an exciting research direction. Closely related is sensitivity analysis, to analyze the robustness of a solution to small performance deviations from expected values, which are inevitable during implementation and integration.

List of Figures

List of Tables

Bibliography

[1] Y. Abdeddaim and O. Maler. Job-shop scheduling using timed automata. In *Computer Aided Verification*, pages 478–492, 2001.

[2] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] ARM. *AMBA Bus*. http://www.arm.com/products/solutions/AMBAOverview.html.

[4] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atalanta, 1991.

[5] Axys Inc. *MaxSim*. http://www.axysdesign.com/products/products_maxsim.asp.

[6] F. Balarin et al. *Hardware-Software Co-Design of Embedded Systems. The POLIS Approach*. Kluwer Academic Publishers, 1997.

[7] S. K. Baruah, D. Chen, and A. K. Mok. Static-priority scheduling of multiframe tasks. In *Euromicro Conference on Real-Time Systems*, June 1999.

[8] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proc. of the IEEE*, 79(9):1270–1282, 1991.

[9] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 2002.

[10] P. Bjureus and A. Jantsch. Heterogeneous system-level cosimulation with SDL and Matlab. In *Proc. 2nd Forum on Design Languages, FDL*, September 1999.

[11] Bosch. *Electronic Control Units*. http://www.kraftfahrzeugtechnik-heute.de:8090/k/en/start/product.jsp?mfacKey=ae_01_stg.

[12] J. T. Buck. Scheduling dynamic dataflow graphs with bounded memory using the Token Flow Model. Technical Report UCB/ERL 93/69, Ph.D dissertation, Dept. of EECS, UC Berkeley, Berkeley, CA 94720, U.S.A., 1993.

[13] G. Buttazzo. *Real-Time Computing Systems - Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 2002.

[14] G. Buttazzo. Rate Monotonic vs. EDF: Judgment day. In *Proc. 3rd Workshop on Embedded Software (EMSOFT)*, Philadelphia (PA), USA, October 2003.

[15] Beatriz Asensio Calvo. Real-time analysis of time-driven scheduling – a quantitative comparison of Round Robin and TDMA. Technical report, Institute of Computer and Communication Networks Engineering, Technical University of Braunschweig, September 2001.

[16] S. Chakraborty, S. Künzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Proc. DATE'03*, Munich, Germany, March 2003.

[17] D. Ching, P. Schaumont, and I. Verbauwhede. Integrated modeling and generation of a reconfigurable network-on-chip. In *2004 Reconfigurable Architectures Workshop (RAW 2004)*, April 2004.

[18] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. A formal methodology for hardware /software codesign of embedded systems. *IEEE Micro*, August 1994.

[19] P. Chou, K. Hines, K. Partidge, and G. Borriello. Control generation for embedded systems based on composition of modal processes. In *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, 1998.

[20] P. Chou, R. B. Ortega, and G Borriello. The Chinook hardware/software co-synthesis system. In *Proc. of the 8th International Symposium on System Synthesis*, Cannes, France, September 1995.

[21] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[22] E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venter, D. Weil, and S. Yovine. TAXYS: A tool for the development and verification of real-time embedded systems. *Lecture Notes in Computer Science*, 2102, 2001.

[23] R. L. Cruz. A calculus for network delay. *IEEE Transactions on Information Theory*, 37(1):114–141, January 1991.

[24] A. Dasdan, D. Ramanathan, and R. K. Gupta. Rate derivation and its applications to reactive, real-time embedded systems. In *Proceedings of the 35th design automation conference*, 1998.

[25] A. Dasdan, D. Ramanathan, and R. K. Gupta. A timing-driven design and validation methodology for embedded real-time systems. *ACM Trans. on Design Automation of Electronic Systems*, 3(4), October 1998.

[26] dSPACE GmbH. *TargetLink 2.0*. http://www.dspace.de/ww/en/pub/products/prodover/targetli.htm.

[27] M. Engels, G. Bilsen, R. Lauwereins, and J. Peperstraete. Cyclo-Static Data Flow: Model and implementation. In *Proc. 28th Asilomar Conf. on Signals, Systems, and Computers*, pages 503–507, Pacific Grove, CA, 1994.

[28] R. Ernst. Codesign of embedded systems: Status and trends. *IEEE Design & Test of Computers*, April 1998.

[29] ETAS. *ERCOSEK Automotive Real-Time Operating System.* http://www.etas.info/html/products/ec/ercosek/en_products_ec_ercosek_index.php.

[30] A. Fehnker. Scheduling a Steel Plant with Timed Automata. In *Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, 1999.

[31] E. Fersman. *A Generic Approach to Schedulability Analysis of Real-Time Systems.* PhD thesis, Uppsala Faculty of Science and Technology, 2003.

[32] C. J. Fidge. Real-time schedulability tests for preemptive multitasking. *Real-Time Systems*, 14:61–93, 1998.

[33] FlexRay Consortium. *FlexRay.* http://www.flexray.com.

[34] D. D. Gajski, J. Zhu, R. Doemer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology.* Kluwer Accademic Publishers, March 2000.

[35] K. Gresser. *Echtzeitnachweis ereignisgesteuerter Realzeitsysteme.* PhD thesis, Technische Universität München, 1993.

[36] K. Gresser. An event model for deadline verification of hard real-time systems. In *Proceedings 5th Euromicro Workshop on Real-Time Systems*, pages 118–123, Oulu, Finland, 1993.

[37] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Publishers, 1993.

[38] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[39] S. Heithecker, A. Lucas, and R. Ernst. A mixed qos sdram controller for fpga-based high-end image processing. In *IEEE Workshop on Signal Processing Systems (SIPS'03)*, Seoul, Korea, August 2003.

[40] R. Henia. Analyse von kontextabhängigem systemverhalten komplexer eingebetteter systeme. Master's thesis, Institute of Computer and Communication Networks Engineering, Technical University of Braunschweig, 2003.

[41] M. Y. Houri. Task graph analysis with complex dependencies. Master's thesis, Institute of Computer and Communication Networks Engineering, Technical University of Braunschweig, 2004.

[42] I-Logix. *Statemate MAGNUM.* http://www.ilogix.com/fs_prod.htm.

[43] IEEE. *VME bus.* http://www.interfacebus.com/Design_Connector_VME.html.

[44] Institute of Computer and Communication Network Engineering, TU Braunschweig. *SPI - System Property Intervals.* http://www.ida.ing.tu-bs.de/research/projects/spi/home.e.shtml.

[45] ITU. *ITU-T. Recommendation Z.100. Specification and Description Language*, 1993.

[46] Kevin Jeffay and Steve Goddard. A theory of rate-based execution. In *IEEE Real-Time Systems Symposium*, 1999.

[47] M. Jersak, Y. Cai, and A. Lucas. Translating simulink to SPI. Technical report, Institute of Computer and Communication Networks Engineering, Technical University of Braunschweig, 2001.

[48] M. Jersak, Y. Cai, D. Ziegenbein, and R. Ernst. A transformational approach to constraint relaxation of a time-driven simulation model. In *Proceedings 13th International Symposium on System Synthesis*, Madrid, Spain, September 2000.

[49] M. Jersak and R. Ernst. Enabling scheduling analysis of heterogeneous systems with multi-rate data dependencies and rate intervals. In *Proceeding 40th Design Automation Conference*, Annaheim, USA, June 2003.

[50] M. Jersak, R. Henia, and R. Ernst. Context-aware performance analysis for efficient embedded system design. In *Proceeding Design Automation and Test in Europe*, Paris, France, March 2004.

[51] M. Jersak, K. Richter, and R. Ernst. Performance analysis for complex embedded applications. *International Journal of Embedded Systems, Special Issue on Codesign for SoC*, 2004.

[52] M. Jersak, K. Richter, R. Ernst, J.-C. Braam, Z.-Y. Jiang, and F. Wolf. Formal methods for integration of automotive software. In *Proc. of Design, Automation and Test in Europe Conference (DATE'03)*, Munich, Germany, March 2003.

[53] M. Jersak, K. Richter, R. Henia, R. Ernst, and F. Slomka. Transformation of SDL specifications for system-level timing analysis. In *Tenth International Symposium on Hardware/Software Codesign (CODES'02)*, Estes Park, Colorado, USA, May 2002.

[54] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[55] G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 74*, pages 471 – 475. North-Holland, 1974.

[56] K. Keutzer, S. Malik, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, 2000.

[57] H. Kopetz and G. Gruensteidl. TTP - a time-triggered protocol for fault-tolerant computin g. In *Proceedings 23rd International Symposium on Fault-Tolerant Co mputing*, 1993.

[58] L.M. Kristensen, S. Christensen, and K. Jensen. The practitioner's guide to coloured petri nets. *International Journal on Software Tools for Technology Transfer*, 2:98–132, 1998.

[59] C.-G. Lee, K. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Bounding cache-related preemption delay for real-time systems. *IEEE Transactions on software engineering*, 27(9):805–826, November 2001.

[60] E. A. Lee et al. Overview of the ptolemy project. Technical report, University of California at Berkeley, March 2001.

[61] E. A. Lee and D. G. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[62] E. A. Lee and D.G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1), January 1987.

[63] E. A. Lee and Th. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, May 1995.

[64] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Trans. CAD*, December 1998.

[65] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings Real-Time Systems Symposiom*, pages 201–209, 1990.

[66] J. Lemieux. *Programming in the OSEK/VDX Environment*. CMP Books, 2001.

[67] LIN Consortium. *LIN - Local Interconnect Network*. http://www.lin-subbus.org/.

[68] C. L. Liu and J. W. Layland. Scheduling algorithm for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20, 1973.

[69] LiveDevices Inc. *Realogy Real-Time Architect Overview*. http://www.livedevices.com/realtime.shtml.

[70] K. Scott M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Pub Co, 2 edition, 1999.

[71] A. Mader. Experiment during Dagstuhl workshop on software intensive embedded systems with special emphasis on automotive, November 2003.

[72] A. Mathur, A. Dasdan, and R. K. Gupta. Rate analysis for embedded systems. *ACM Trans. on Design Automation of Electronic Systems*, 3(3), July 1998.

[73] The MathWorks. *Matlab*. http://www.mathworks.com.

[74] The MathWorks. *Simulink*. http://www.mathworks.com.

[75] The MathWorks. *Stateflow*. http://www.mathworks.com.

[76] The MathWorks, Inc. *Real-Time Workshop, version 5*. http://www.mathworks.com/access/helpdesk/help/toolbox/rtw/rtw.shtml.

[77] A. Maxiaguine, S. Künzli, and L. Thiele. Workload characterization model for tasks with variable execution demand. In *Proc. Design, Automation and Test in Europe (DATE 2004)*, Paris, France, 2004.

[78] Mentor Graphics. *Seamless*. http://www.mentor.com/seamless/.

[79] A.K. Mok and D. Chen. A multiframe model for real-time tasks. *IEEE Transactions on Software Engineering*, 23(10):635–645, 1997.

[80] MOST Cooperation. *MOST - Media Oriented Systems Transport*. http://www.mostcooperation.com/.

[81] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. In *CODES+ISSS'03*, Newport Beach, California, USA, October 2003.

[82] H. Oh and S. Ha. Fractional rate dataflow model and efficient code synthesis for multimedia applications. In *Languages, Compilers, and Tools for Embedded Systems (LCTES'02) and Software and Compilers for Embedded Systems (SCOPES'02)*, Berlin, Germany, June 2002.

[83] Open SystemC Initiative. *SystemC*. http://www.systemc.org/.

[84] J. C. Palencia, J. J. Gutierrez Garcia, and M. G. Harbour. Best-case analysis for improving the worst-case schedulability test for distributed hard real-time systems. In *Proceedings of 10th Euromicro Workshop on Real-Time Systems*, June 1998.

[85] J. C. Palencia and M. G. Harbour. Schedulablilty analysis for tasks with static and dynamic offsets. In *Proc. 19th IEEE Real-Time Systems Symposium (RTSS98)*, Madrid, Spain, 1998.

[86] J. C. Palencia and M. G. Harbour. Exploiting precedence relations in the schedulability analysis of distributed real-time systems. In *Proceedings of the 20th Real-Time Systems Symposium*, December 1999.

[87] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–257, June 1993.

[88] Philips. *Nexperia*. http://www.semiconductors.philips.com/ products/nexperia/index.html.

[89] J. Pino, S. Bhattacharyya, and E. Lee. A hierarchical multiprocessor scheduling system for dsp applications. In *Proc. of IEEE Asilomar Conference on Signals, Systems, and Computers*, November 1995.

[90] P. Pop, P. Eles, and Z. Peng. Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems. In *Tenth International Symposium on Hardware/Software Codesign (CODES'02)*, Estes Park, Colorado, USA, May 2002.

[91] P. Pop, P. Eles, and Z. Peng. Schedulability analysis and optimization for the synthesis of multi-cluster distributed embedded systems. In *Proc. Design, Automation and Test in Europe (DATE 2003)*, Munich, Germany, 2003.

[92] P. Pop, P. Eles, Z. Peng, V. Izosimov, M. Hellring, and O. Bridal. Design optimization of multi-cluster embedded systems for real-time applications. In *Proc. Design, Automation and Test in Europe (DATE 2004)*, Paris, France, 2004.

[93] R. Bosch GmbH. *CAN Specification Version 2.0*, 1991.

[94] Rational Software. *Purify*. http://www-306.ibm.com/software/awdtools/purifyplus/.

[95] K. Richter. Developing a general model for scheduling of mixed transformative/reactive systems. Master's thesis, Technical University of Braunschweig, January 1998.

[96] K. Richter. *Compositional Scheduling Analysis Using Standard Event Models*. PhD thesis, Technical University of Braunschweig, 2004.

[97] K. Richter, M. Jersak, and R. Ernst. A formal approach to MpSoC performance verification. *IEEE Computer*, 36(4), April 2003.

[98] K. Richter, R. Racu, and R. Ernst. Scheduling analysis integration for heterogeneous multiprocessor SoC. In *Proceedings 24th International Real-Time Systems Symposium (RTSS'03)*, Cancun, Mexico, December 2003.

[99] Semiconductor Industry Association. *2001 International Technology Roadmap for Semiconductors*. http://public.itrs.net/Files/2001ITRS/Home.htm.

[100] M. Sgroi, L. Lavagno, and A. Sangiovanni-Vincentelli. Formal models for embedded system design. *IEEE Design & Test of Computers*, 17(2):14–27, 2000.

[101] L. Sha, R. Rajkumar, and S. S. Sathaye. Generalized rate-monotonic scheduling theory: A framework for developing real-time systems. *Proceedings of the IEEE*, 82(1):68–82, January 1994.

[102] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9), September 1990.

[103] F. Slomka, M. Dörfel, and R. Münzenberger. Generating mixed hardware/software systems from SDL specifications. In *Proc. 9th International Symposium on Hardware/Software Co-Design (CODES 2001)*, Copenhagen, Denmark, April 2001.

[104] Sonics. *SiliconBackplane III MicroNetwork IP*. http://www.sonicsinc.com/sonics/products/siliconbackplaneIII/.

[105] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hart real-time systems. In *Journal of Real-Time Systems*, 1989.

[106] Synopsys. *System Studio*. http://www.synopsys.com/products/cocentric_studio/.

[107] Technical University of Braunschweig. *SymTA/S – Symbolic Timing Analysis for Systems*. http://www.symta.org.

[108] Telelogic. *Tau*. http://www.telelogic.com/products/tau/.

[109] L. Thiele, S. Chakraborty, M. Gries, and S. Künzli. Design space exploration of network processor architectures. In Mark Franklin, Patrick Crowley, Haldun Hadimioglu, and Peter Onufryk, editors, *Network Processor Design Issues and Practices, Volume 1*, chapter 4, pages 55–90. Morgan Kaufmann, October 2002.

[110] L. Thiele, S. Chakraborty, M. Gries, and S. Künzli. Design space exploration of network processor architectures. In *First Workshop on Network Processors at the 8th International Symposium on High-Performance Computer Architecture (HPCA8)*, Cambridge MA, USA, February 2002.

[111] TimeSys. *TimeWiz*. http://www.timesys.com/.

[112] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing & Microprogramming*, 50(2-3):117–134, apr 1994.

[113] K. W. Tindell. Adding time-offsets to schedulability analysis. Technical Report YCS 221, Univ. of York, 1994.

[114] K. W. Tindell. An extendible approach for analysing fixed priority hard real-time systems. *Journal of Real-Time Systems*, 6(2):133–152, Mar 1994.

[115] Ken Tindell, Herman Kopetz, Fabian Wolf, and Rolf Ernst. Safe automotive software development. In *Proc. Design, Automation and Test in Europe (DATE'03)*, Munich, Germany, March 2003.

[116] Tri-Pacific Software, Inc. *RAPID RMA*. http://www.tripac.com/html/prodfact-rrm.html.

[117] University of California at Berkeley. *Ptolemy*. http://ptolemy.eecs.berkeley.edu/ptolemyII.

[118] Uppsala Universitet and Aalborg University. *UPPAAL*. http://www.uppaal.com/.

[119] WindRiver. *VxWorks*. http://www.windriver.com/announces/vxworks/.

[120] F. Wolf. *Behavioral Intervals in Embedded Software*. Kluwer Academic Publishers, 2002.

[121] F. Wolf, R. Ernst, and W. Ye. Path clustering in software timing analysis. *IEEE Transactions on VLSI Systems*, 9(6), December 2001.

[122] T. Yen and W. Wolf. Performance estimation for real-time distributed embedded systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(11), November 1998.

[123] D. Ziegenbein. *A Compositional Approach to Embedded System Design*. PhD thesis, Technical University of Braunschweig, 2003.

[124] D. Ziegenbein, R. Ernst, K. Richter, J. Teich, and L. Thiele. Combining multiple models of computation for scheduling and allocation. In *Proceedings Sixth International Workshop on Hardware/Software Co-Design (Codes/CASHE '98)*, pages 9–13, Seattle, USA, March 1998.

[125] D. Ziegenbein, M. Jersak, K. Richter, and R. Ernst. Breaking down complexity for reliable system-level timing validation. In *Ninth IEEE/DATC Electronic Design Processes Workshop (EDP'02)*, Monterey, USA, April 2002.

[126] D. Ziegenbein, K. Richter, R. Ernst, L. Thiele, and J. Teich. SPI – A system model for heterogeneously specified embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10(4), August 2002.

[127] D. Ziegenbein, J. Uerpmann, and R. Ernst. Dynamic Response Time Optimization for SDF Graphs. In *Proceedings International Conference on Computer-Aided Design (ICCAD '00)*, San Jose, November 2000.